前　言

数据结构是计算机程序设计重要的理论技术基础，它不仅是计算机学科的核心课程，而且已经成为计算机相关专业必要的选修课。其要求是学会分析、研究计算机加工的数据结构的特性，初步掌握算法的时间和空间分析技术，并能够编写出结构清晰、正确易读的算法，达到培养数据抽象能力的目的。学习数据结构可以使读者碰到具体问题时，能够找到一个优化的存储结构和解决方法。本书利用目前流行的开发工具 Java 语言进行数据结构设计，包含了数据结构的全部内容，符合大学的教学大纲，既可以作为大学数据结构课程的教材，又可以为程序设计者学习数据结构提供帮助。

本书以数据结构为主线，是在 Java 语言的基础之上编写的，希望读者在阅读本书之前，最好具备 Java 语言基础。这样，在学习数据结构时，能够比较容易地建立正确的数据结构中的存储和逻辑概念。

本书共分 10 章，第 1 章综述了数据结构中的基本概念；第 2 章主要描述了线性结构的存储与实现；第 3 章描述了特殊的线性结构的存储及其实现；第 4 章着重描述了数组的存储及数组的运算；第 5 章描述了层次结构的各种运算；第 6 章描述了网状结构的存储及实现算法；第 7 章介绍了各种排序的方法及算法比较；第 8 章主要介绍了查找方法；第 9 章介绍了操作系统中涉及的动态存储管理的基本技术；第 10 章介绍了常用文件结构。本书的内容突出了抽象数据类型的概念，对每一种数据结构都给出了相应的抽象数据类型的规范说明和实现。

我们向使用本教材的教师免费提供本书的电子教案，其下载网址为 http://www.tupwk.com.cn/downpage/index.asp。需要本书习题参考答案的教师请发邮件至 cwkbook@tup.tsinghua.edu.cn，邮件的主题请设为“获取《数据结构与算法分析》参考答案”。

本书的第 1~4 章、第 9 和第 10 章由王世民编写，第 5 和第 6 章由朱建方编写，第 7 和第 8 章由孔凡航编写，疏漏之处敬请读者提出宝贵意见或建议。

作　者

目　　录

第1章　数据结构概论

信息是计算机科学的基础，信息必须以数据的方式，按照一定的规则存储在计算机的存储器中。对数据的操作方法如增加、插入、删除、查询等既要考虑数据的存储，又要考虑数据操作中数据的处理速度等各方面的因素。学习数据结构及其相关算法可以有效地完成数据处理并提高数据的处理效率。

本章的学习目标：

- 数据结构的研究内容；
- 数据结构中的基本概念、常用术语；
- 学习数据结构的意义；
- Java 语言描述；
- 对算法进行描述和分析的方法。

1.1　什么是数据结构

众所周知，计算机是一种信息处理装置，信息中的各个数据元素并不是孤立存在的，它们之间存在着一定的结构关系。如何表示这些结构关系，如何在计算机中存储数据和信息，采用什么样的方法和技巧加工与处理这些数据，都是数据结构课程需要努力解决的问题。

一般来说，使用计算机解决具体问题时，通常需要几个步骤：分析具体问题得到数学模型，设计解决数学模型的算法，编制程序并调试，最后得到最终答案。

要想得到数据模型，必须了解数据之间的关系，在数据结构中数据之间的关系主要有两种，它们分别是线性关系和非线性关系，其中非线性关系又可以分为树型关系和图关系。确定了数据之间的关系后，可以将数据存储在计算机内，所以说数据的逻辑结构和存储结构是密不可分的两个方面，在实现算法时，首先应解决数据的存储问题。

为了说明这些，可以通过以下的例子说明。

例 1：图 1-1 是一个班级若干名学生的登记表，使用计算机管理该表格，可以把所有学生的登记表看作一个文件，它由若干条记录组成，每个学生对应一条记录，每条记录占一行，每行中有若干列，包括学号、姓名、性别、年龄等属性，反应了每个学生的基本情况，每条记录按学号从小到大排列。

学号	姓名	性别	年龄	民族	……
001	刘晓东	男	18	汉	……
002	张立	男	19	汉	……
003	韩娟	女	18	汉	……
……	……	……	……	……	……

图 1-1　线性结构图

因为每个学生的属性相同，将每个学生的各种属性集合抽象为一个独立的数据单位，与每个数据单位相邻的前一个数据单位最多只能有一个(第一个数据单位没有)，与每个数据单位相邻的后一个数据单位最多只能有一个(最后一个数据单位没有)。这时可以将文件中的数据单位的集合称为数据集合，我们将这种数据单位之间的关系称为线性关系。也就是说，该数据文件的逻辑结构为线性结构，也称该文件为一个线性表。在这种结构中，计算机内可以采用多种方法存储，在存储数据时，只要能够体现出数据元素之间的线性关系即可。

对于这个文件，我们可以进行的操作有：如果某个学生离开学校，应从该文件中将此学生删除，对应的操作就是删除一个数据元素；如果新来一位同学，则相应的操作是在该线性表中插入一个数据元素；新年伊始，每位同学的年龄应增加一岁，对应的操作就是修改所有同学的年龄；如果不要求按学号从小到大排列，需要按姓名的字典顺序排列，对应的操作就是排序等。

所以数据之间既要考虑存储，又要考虑数据单位之间的关系，在确定了存储结构后，根据存储的结构再来确定相应操作的实现方法。

例 2：我们使用的微机中，每个逻辑盘都有一个根目录，根目录下可以建立若干个子目录，而每个子目录下又可以创建若干个子目录(如图 1-2 所示)，形成一个对应多个的关系称为层次关系(树型关系)。

如果将每个目录作为一个独立的数据单位，那么每个子目录只有一个父目录，但是它又允许有若干个子目录。我们可以将目录的集合看作是数据的集合，目录之间的关系看作是非线性关系中的层次模型。

在此例中，比较典型的是根目录不能删除，每个子目录可以删除，并且每个目录下可以存储若干文件。此时的存储像一棵倒树，根目录像大树的树根(只有一个)，子目录像大树的树枝(允许多个)，而每个目录下的文件又像树枝上的树叶，此种结构层次分明，所以又称层次结构。

对应此种结构的操作有：如果某个目录需要删除(根目录例外)，对应的操作就是删除该目录以及该目录下的文件，又称删除结点(子树)；如果在此结构下增加一个管理目录，对应的操作是在树结构中添加一个结点；查找某个子目录，我们一般的操作是从根目录开始，依次往下查找，对应的数据结构的操作就是查找等。

在此种结构中比较关键的是各种对应的操作一般从根目录开始，对应数据结构中的操作就是从根结点开始。

例 3：如果以每台计算机为结点，组成计算机网络，在计算机网络中，每台计算机之

间可以实现通信，此时每台计算机之间可以与多台计算机进行通信(如图 1-3 所示)。

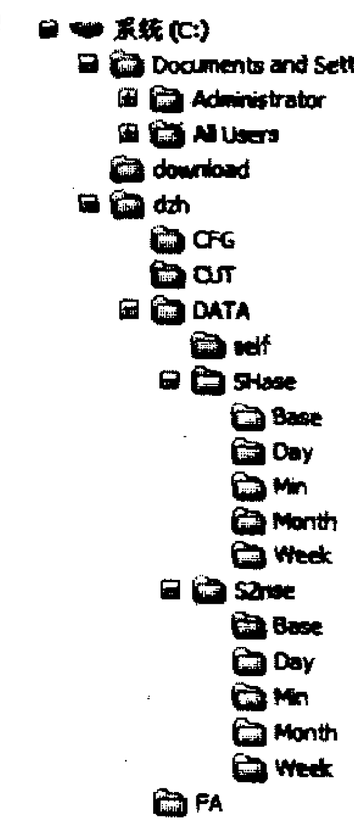


图 1-2　层次结构图

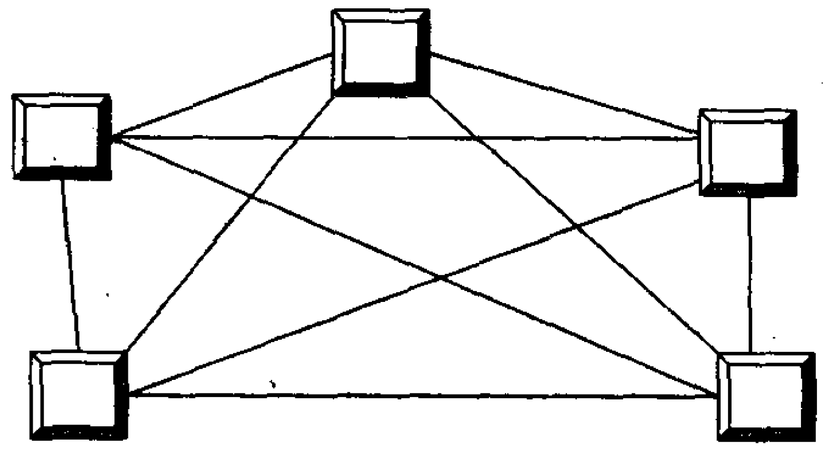


图 1-3　图结构

如果以每台计算机为结点，每台计算机可以通过传输介质与其他计算机进行通信，此时每台计算机之前可以有多台计算机，每台计算机之后也可以有多台计算机。简单将计算机之间的关系可以理解为多对多的关系，属于非线性关系中的图关系。

在此种结构中以每台计算机作为结点，如果它为其他计算机提供服务，则该计算机就是被提供服务的计算机的前趋，接受服务的计算机则称为该计算机后继；同时为其他计算机提供服务的计算机可能又接受多台计算机为它提供的服务，所以说每台计算机既是多台计算机的前趋，又是多台计算机的后继。

如果将此结构存储后，所对应的操作有：如果在网络中增加一台计算机，对应的操作是添加一个结点；如果在此结构中删除一个计算机用户，对应的操作是删除一个结点；在网络中搜寻一台计算机，对应的操作是查询等。

将以上的 3 个例子简单总结，可以说在处理数学模型时，首先应该保存数学模型所需要的必须的数据，根据数据在计算机中的存储，以及各个独立的数据元素之间的关系，可以将数据结构分为线性结构和非线性结构。

在客观世界中，如何体现数据之间的关系，需要计算机专业人员对数据进行分析时分类总结，以便更加深入地了解各种结构的特性以及关系。从不同的角度对数据结构进行分类，最终的目的就是更好地认识各种结构的特性及关系，而在设计某个操作时，首先应该清楚数据元素之间具有的逻辑关系，它适合什么样的存储结构来具体实现这种操作。同时同一种操作在不同的存储结构中实现的方法可以不同，而有的实现则完全依赖于所采用的存储结构，所以研究数据之间的关系直接影响了问题的求解，由此可见数据结构的重要性。

简单地说，数据结构是研究数据的存储、数据之间的关系及对数据实现各种操作的一门学科。

数据结构的定义可以记作：

Data-Structure=(D,R)

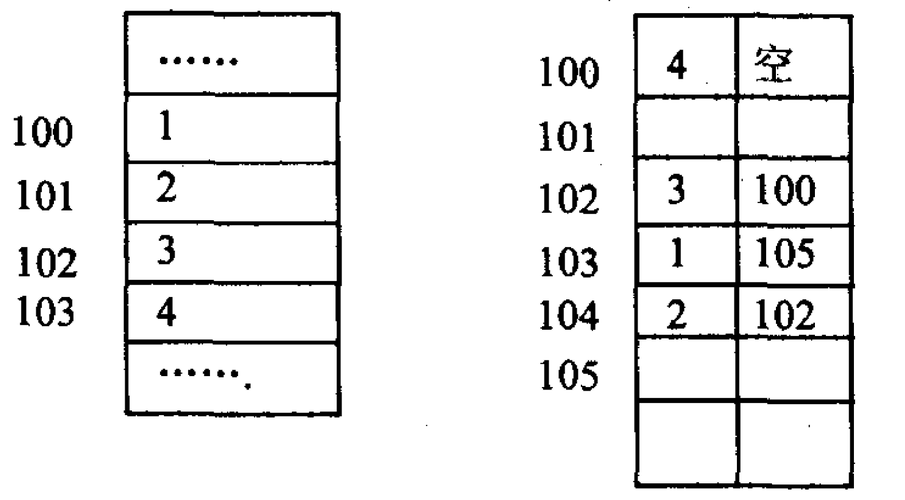
其中，D 是数据元素的有限集合，R 是 D 上的关系。

一般情况下，“关系”是指数据元素之间存在的逻辑关系，也称为数据的逻辑结构。数据在计算机内的存储表示(或映像)称为数据的存储结构或物理结构。

逻辑结构体现的是数据元素之间的逻辑关系，换句话说就是从操作对象中抽象出来的数学模型，因此又称为抽象结构，通常我们习惯说的数据结构一般就是指的逻辑结构。然而讨论数据结构的目的是为了在计算机中实现对数据的操作，因此还需要研究数据的存储结构。

存储结构是数据在计算机内的表示(映像)，又称物理结构。它包括数据元素的表示和关系的表示。由于映像的方法不同，所以同一种逻辑结构可以映像成两种不同的存储结构：顺序映像(顺序存储结构)和非顺序映像(非顺序存储结构)。顺序映像的特点是在顺序存储结构(一般用一维数组)中体现数据之间的关系；而非顺序存储结构则一般采用指针实现数据之间的关系，包括链式存储结构(链表)和散列结构等。数据的存储结构要能够正确反映数据元素之间的逻辑关系。也就是说，数据的逻辑结构和数据的存储结构是密不可分的两个方面，任何一种算法的设计取决于选定的逻辑结构，而算法的实现则依赖于采用的存储结构。

顺序映像(顺序存储结构)是借助元素在存储器中位置表示数据元素之间的逻辑关系，或逻辑上相邻的结点存储在物理位置上相邻的存储单元里，结点的逻辑关系由存储单元的邻接关系来体现；而非顺序映像(链式存储结构)是借助元素存储地址的指针表示元素之间的逻辑关系，或逻辑上相邻的结点在物理位置上可相邻，可不相邻，逻辑关系由附加的指针段表示。例如图 1-4 所示。



(a) 顺序映像示意图　　(b) 非顺序映像示意图

图 1-4　顺序存储和非顺序存储逻辑图

数据的非顺序存储结构除包括链式存储结构(简称链表)以外，还有散列存储结构、索引存储结构等。

链式存储结构利用指针直接表示数据元素之间的关系。

散列结构的基本思想是根据结点的关键字，利用散列函数直接计算出该结点的存储地址。

索引存储结构是指在存储结点信息的同时，还建立附加的索引表。索引表的每一项称为索引项，索引项的一般形式是：(关键字，地址)。关键字是能够惟一标识一个结点的那些数据项集合。索引存储结构分为稠密索引和稀疏索引，其中稠密索引是指每个结点在索引表中都有一个索引项的索引表；而稀疏索引是指一组结点在索引表中对应一个索引项的索引表。

针对存储结构，数据元素存储在计算机中，应对每个数据元素确定其取值范围以及在值上定义的一组操作就是数据类型。数据类型是和数据结构密切相关的一个概念，用以刻画(程序)操作对象的特征。在用高级语言编写的程序中，每个变量、常量或表达式都有一个它所属的数据类型，类型明显或隐含地规定了在程序执行期间变量或表达式所有可能的取值范围，以及在这些值上允许执行的操作。因此数据类型是指一个值的集合以及在这些值上定义的一组操作的总称。例如 Java 语言中有整数类型、字符类型、逻辑类型等。

数据类型根据是否允许分解分为原子类型和结构类型，其中原子类型是指其值不可再分的数据类型，例如整型、字符型等；而结构类型是指其值可以再分解为若干成分(分量)的数据类型，例如数组的值由若干分量组成。

实际上计算机中数据类型的概念不仅局限在高级语言中，从计算机和程序设计用户的角度来说，“位”、“字节”、“字”是计算机硬件的原子类型，程序设计用户中使用的高级语言提供的数据类型需要编译或解释转化为更低级语言(汇编或机器语言)的数据类型来实现。这样对使用者来说，不必了解数据类型在计算机内的表示，也不必了解数据类型的操作实现，实现了信息的隐蔽，方便了用户使用。

根据数据的结构(逻辑结构和存储结构)特性在数据的生存期间的变动情况，可将数据结构分为静态结构和动态结构。静态结构是指在数据存在期不发生任何变动，例如高级语言中的静态数组；而动态结构是指在一定范围内结构的大小可以发生变动，如使用的堆栈。

总之，数据结构所要研究的主要内容简单归纳为以下 3 个方面：

- 研究数据元素之间的客观联系(逻辑结构)；
- 研究数据在计算机内部的存储方法(存储结构)；
- 研究如何在数据的各种结构(逻辑的和物理的)上实施有效的操作或处理(算法)。

所以数据结构是一门抽象地研究数据之间的关系的学科。

1.2　数据结构的发展史及其在计算机科学中的地位

数据结构作为一门独立的课程始于 1968 年，在我国，数据结构作为一门独立课程是在 20 世纪 80 年代初，早期的数据结构对课程的范围没有明确的规定，数据结构的内容几乎和图论、树的理论是相同的，在 20 世纪 60~70 年代，随着大型程序的出现，软件也相对独立，结构程序设计逐步成为程序设计方法学的主要内容，人们已经认识到程序设计的实质就是对所确定的问题选择一种好的结构，从而设计一种好的算法。

目前在我国的高等教育体系中，数据结构不仅作为计算机专业教学计划中的重点核心课程之一，而且也开始成为许多非计算机专业的主要选修课程。在计算机专业中它是在程序设计之后的各专业课程之前的一门最重要的专业基础课，它为从事各类系统软件和应用软件的设计提供了必备的知识和方法。

数据结构在计算机科学领域中有着十分重要的地位，它有自己的理论、研究对象和应

用范围，它的内容也随着计算机技术的飞速发展而不断地扩充，从包括网络、集合代数论等离散结构的内容到包括数据库的内容等。

数据结构课程一般的前序课程是离散数学和程序设计基础，后序课程有操作系统、编译原理、数据库原理等。数据结构的研究不仅涉及到计算机硬件(特别是编码理论、存储装置和存取方法等)的研究范围，而且和计算机软件有着更密切的关系，无论是编译原理还是操作系统，都涉及数据元素在存储器中的分配问题。即使在研究信息检索时也必须考虑如何组织数据，以便查找和存取数据元素更为方便。因此数据结构是介于数学、计算机硬件和计算机软件之间的一门核心课程(如图 1-5 所示)。

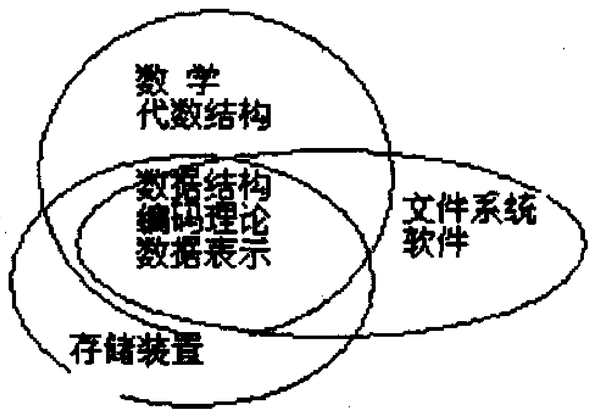


图 1-5　数据结构所处的位置

值得注意的是，数据结构的发展并未终结，一方面，面向专门领域中的特殊问题的数据结构得到研究和发展，如多维图形数据结构等；另一方面，从抽象数据类型观念讨论数据结构，已越来越被人们所重视。

1.3　基本概念和术语

数据在计算机科学中是指输入到计算机中并能够被计算机识别、存储和加工处理的符号的总称，例如一个整数或一个实数。对计算机科学而言，数据的定义非常广泛，计算机所能接受的所有符号，如我们说话，写的字，日常生活中看到的图像等，都可以通过编码的方式存储在计算机中，所以它们统称为数据结构中的数据。

数据元素是数据的基本单位，通常把数据元素作为一个整体进行考虑和处理。例如文件中的记录可以看作是一个整体。数据元素通常可以理解为逻辑意义上的数据的基本单位，而不是物理意义上的基本数据单位。

数据元素可由一个或多个数据项组成。也就是说，数据的基本物理单位是数据项。数据项(或数据元素)是指具有独立含义的最小识别单位(数据中不可分割的最小单位)。例如文件记录中的某一列，所以数据项又称项或字段。

数据项有逻辑形式(logical form)和物理形式(physical form)两个方面。用 ADT 给出的数据项的定义是它的逻辑形式，数据结构中对数据项的实现是它的物理形式。

结构通常是指关系，所以数据结构是指数据之间的关系的一门学科，它表示的是数据之间的相互形式，即数据的组织形式。数据结构可以从不同的角度进行分类，一般从用户的角度和计算机的角度来划分数据结构的分类：从用户的角度来说，主要关心的是数据元素之间的关系，所以数据结构称为逻辑结构(或抽象结构)，其作用是研究数据元素之间的逻辑(或抽象)关系；而从计算机的角度来说，主要关心的是数据元素及其关系在计算机内如何存储，称为存储结构(或物理结构)，其作用是数据元素及其关系在计算机内的表示。

如果从数据结构是否变化来分类，可以把数据结构划分为静态结构和动态结构。所谓静态结构是指数据结构在存在期间不会发生变动，如静态数组，不会随着操作的进行而改变数组的大小。而动态结构是指在一定范围内结构的大小要发生变动，如堆栈、队列以及树型结构等都属于动态结构的范畴。

根据数据结构是研究数据及其关系的一门学科的定义，了解数据是对客观事物的符号表示是非常必要的。数据在计算机中存储必须按照数据的分类存储，数据分类应该用数据类型划分，数据类型用以刻画(程序)操作对象的特性，在高级语言编写的程序中，每个变量或表达式都有一个它所属的确定的数据类型，类型明显或隐含地规定了在程序执行期间变量或表达式所有可能的取值范围，以及在这些值上允许进行的操作。因此数据类型是一个值的集合和定义在这个值集合上的一组操作的总称。如程序语言中的整型变量、字符型变量，都确定了数据的取值范围。

按照值的不同属性，数据类型可以分为两类：原子类型和结构类型。原子类型的值是不可分解的，如高级语言中基本数据类型(整型、字符型等)；结构类型是由若干成分按某种结构组成的，是可以分解的。

实际上，在计算机中，数据类型的概念并非局限在高级语言中，计算机的硬件和软件系统都提供了一组原子类型或结构类型，例如“位”、“字节”、“字”等属于原子类型，它们的操作通过计算机设计的指令直接由电路系统完成，而高级语言提供的数据类型则是通过编译或解释器转化为低层(汇编语言或机器语言)的数据类型来实现。

在存储结构中包括了数据元素的表示和数据元素之间的关系表示。数据元素存储在计算机中，计算机中表示信息的最小单位是二进制数的一位，称为位(bit)，由若干位组成一个位串表示一个数据元素，称为元素或结点，当数据元素由若干个数据项组成时，对应于各个数据项的子位串称为数据域。在非顺序映像中借助指示元素存储地址，描述数据元素之间逻辑关系的部分称为指针域。结点也可以描述为数据处理的数据单位，它可能是一条记录、一个数据项或组合数据项。对应结点定义根据结点所处位置的不同可以将表中的结点分为前趋和后继结点，对表中任意结点，处于该结点之前的所有结点称为该结点的前趋结点，而处于该结点之后的所有结点称为该结点的后继结点，与之相邻的前趋结点称为直接前趋结点，而与之相邻的后继结点称为直接后继结点。表中的第一个结点称为开始结点，表中最后一个没有后继的结点称为终端结点。

1.4　抽象数据类型和数据结构

数据类型是指一个值的集合以及在这些值上定义的一组操作的总称。抽象数据类型(Abstract Data Type，ADT)是指抽象数据组织和与之相关的操作。每一个操作由它的输入和输出定义。抽象数据类型的定义取决于它的一组逻辑特性，而与其在计算机内的表示和实现无关，也就是说，无论其内部结构如何变化，只要其数学特性不变，都不影响其外部

的使用。或者说一个 ADT 的定义不涉及它的实现细节，这些实现细节对 ADT 的用户是隐藏的。隐藏实现细节的过程称为封装。数据结构是 ADT 的物理实现。

抽象数据类型和数据类型实质上是一个概念。例如计算机中的“整数”类型是一个抽象类型，尽管它们在不同的处理器上实现的方法不同，但由于其定义的数学特性相同，在用户看来都是相同的。因此抽象的意义在于数据类型的数学抽象特性。

抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关，即不论其内部结构如何变化，只要其数学特性不变，都不影响其外部的使用。

另一方面，抽象数据类型比数据类型的范畴更广，它不仅局限在处理器中已经定义并实现的数据类型，还包括用户在设计软件时自己定义的数据类型。一个软件系统的框架应建立在数据之上，而不应该建立在操作之上。所以定义的数据类型的抽象的层次越高，含有该抽象数据类型的软件模块的复用程度也就越高。

抽象数据类型可以定义为：

$$(D,S,P)$$

其中，D 表示数据对象，S 是 D 上的关系集，P 是对 D 的基本操作集。

ADT 使用伪码描述为：

```
ADT 抽象数据类型名{
    数据对象:<数据对象的定义>
    数据关系:<数据关系的定义>
    基本操作:<基本操作的定义>
}ADT 抽象数据类型名
```

抽象数据类型的定义由一个值域和定义在该值域上的一组操作组成，如果按照其值的不同特性，可细分为 3 种类型。

- 原子类型：属于原子类型的变量的值是不可再分的。
- 固定聚合类型：属于该类型的变量的值由确定数目的成分按照某种结构组成。
- 可变聚合类型：属于该类型的变量的值的成分数目不确定，其中序列的长度是可变的。

后两种类型可统称为结构类型。

例 1：整数的数学概念和施加到整数的运算构成一个 ADT。Java 的变量类型 int 就是对这个抽象类型的物理实现。但 int 型变量有一定的取值范围，所以对这个抽象的整型的实现并不完全正确，如果无法接受该限制，就必须引进一些其他的实现方法来实现这个抽象类型。

例 2：一个整数线性表的 ADT 应包含下列操作。

- 数据的存储结构确定线性关系。
- 把一个新整数插入到表尾。
- 按线性表的元素顺序依次打印。

- 删除线性表的某个元素。
- 查找线性表的某个元素，成功返回 true，失败返回 false。
- 将线性表的整数排序。

通过上述的描述，每个操作的输入和输出清晰可见，但是对整数线性表的实现过程并没有详细说明。

例 3：抽象数据类型的三元组定义。

```
ADT Tri{
   数据对象：D={e1, e2, e3| e1, e2, e3∈elementset(已经定义的数据集合)}
   数据关系：R1={< e1  e2>   R2= < e2, e3>}
   基本操作：
    inittri(&t,v1,v2,v3)
    结果：构造三元组 t，并将 v1、v2、v3 赋值给 e1、e2、e3 完成初始化。
   tansporttri(m,&t)
    结果：将矩阵 M 转置为 T。
    addtri(m,n,&q)
    结果：求矩阵 m 和 n 和，结果放在 q 中。
}ADT tri
```

对应的各种操作，用户自己编写出来即可。

ADT 的概念就是将复杂的问题抽象化。抽象化之后的问题是使我们重视问题而忽略不必要的细节。

ADT 不同于类，其区别在于 ADT 相当于在概念层(或为抽象层)上描述问题，而类相当于在实现层上描述问题。

Java 语言作为面向对象的程序设计语言，它提供了许多特性来支持封装。读者只要具备结构化的程序设计语言(如 Pascal 或 C)的编程经验，应该比较容易理解本书的算法实现。

1.5　学习数据结构的意义

数据结构是计算机专业的核心课程之一，在众多的计算机系统软件和应用软件中都用到各种数据结构。

著名的瑞士计算机科学家 N.Wirth 曾提出：算法+数据结构=程序。其中数据结构是指数据逻辑结构和物理结构，算法是对数据运算的描述。由此可见，程序设计的实质是对具体问题选择一种好的数据结构，再设计一个好的算法，而好的算法通常取决于实际问题的数据结构。下面以两个例子加以说明。

例 1：电话号码的查询问题。

假设编写一个程序，查询某个城市或私人的电话号码。对任意给出的一个姓名，若该人有电话号码，则要迅速地找到其电话号码；否则指出该人没有电话号码。

解决此问题首先要构造一张电话号码登记表，表中每个结点至少存放两个数据项：姓名和电话号码。要写出好的查找算法，取决于这张表的结构及存储方式。我们考虑两种方法：

(1) 将表中结点顺序地存储在计算机中。

查找方法是从头开始依次查对姓名，直到找出正确的姓名或是找遍整个表均没有找到为止。这种查找算法对于一个不大的单位或许是可行的，但对一个有成千上万私人电话的城市就不实用了。

(2) 若这张表是按姓氏排列的，则我们可以另造一张姓氏索引表，采用如图 1-6 所示的存储结构。

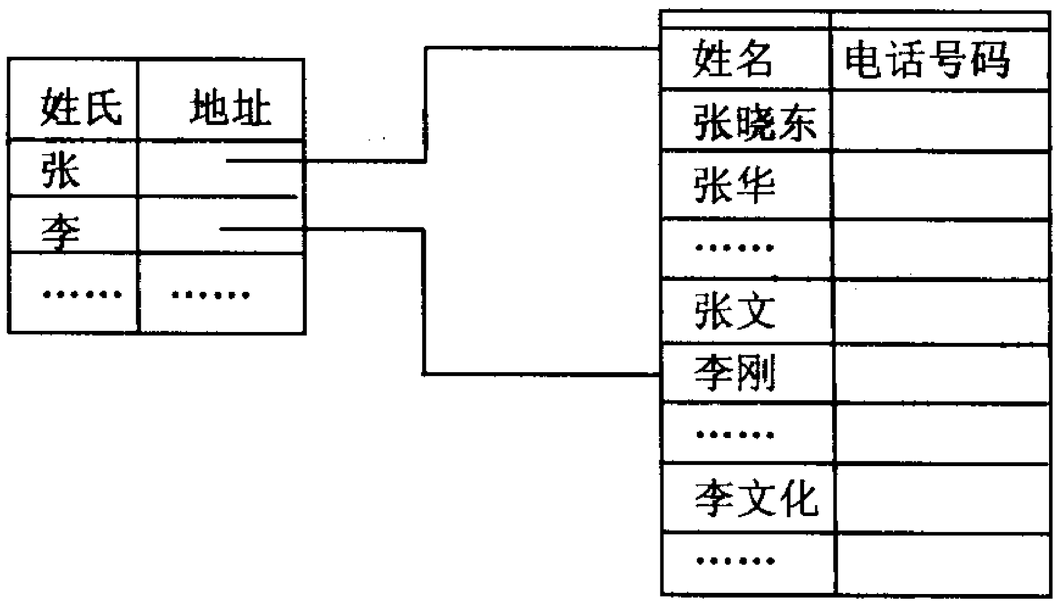


图 1-6　电话号码查询索引存储图

查找过程是先在索引表中查对姓氏，然后根据索引表中的地址到电话号码登记表中核查姓名，这样查找登记表时就无需查找其他姓氏的名字了。因此，在这种新的结构上产生的查找算法就更为有效。

例 2：田径赛的时间安排问题。

假设某校的田径选拔赛共设 6 个项目的比赛，即跳高、跳远、标枪、铅球、100 米和 200 米短跑，规定每个选手至多参加 3 个项目的比赛。现有 5 名选手报名参赛，选手所选择的项目如表 1-1 所示。现在要求设计一个竞赛日程安排表，使得在尽可能短的时间内安排完比赛。

表 1-1　参赛选手比赛项目表

姓　　名	项目 1	项目 2	项目 3
刘晓东	跳高	跳远	100m
马明	标枪	铅球	
张文	标枪	100m	200m
李文化	铅球	200m	跳高
王强	跳远	200m	

为了能较好地解决这个问题，首先应该选择一个合适的数据结构来表示它。为此，根

据表 1-1 我们可以设计这样的一个图(如图 1-7 所示)，其中顶点表示竞赛项目(a、b、c、d、e、f 分别表示 6 种竞赛项目)，显然每个人不可能同时参加两个项目的竞赛，在所有不能同时比赛的项目之间连上一条边。

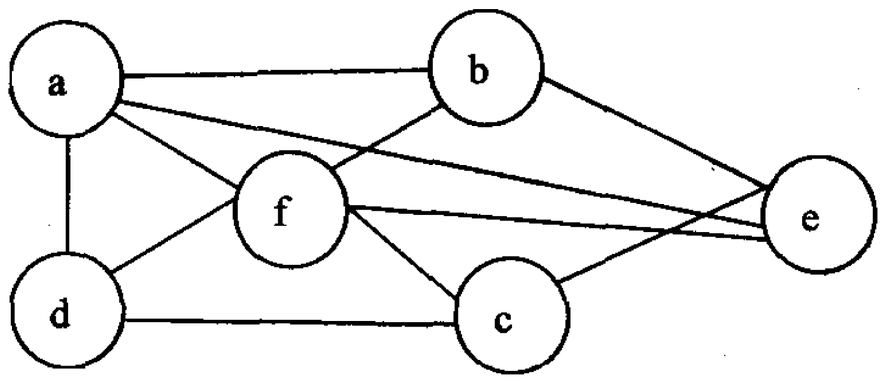


图 1-7　竞赛项目的数据结构模型

假设用每一种颜色表示相同时间，我们可以使用尽可能少的颜色为每个顶点着色，使得任意一条边连接的两个顶点颜色不同，相同颜色的顶点(竞赛项目)可以安排在同一个时间。例如 a 和 c 不相邻，可以是同一种颜色；b 和 d 可以是同一种颜色；e 和 f 相邻，分别用两种颜色。也就是说，总共可以使用 4 种颜色描述。

数据结构包括了逻辑结构、存储结构和算法。解决问题的关键是：选择合适的数据结构表示问题，然后写出有效的算法。

1.6　Java 语言概述

1.6.1　面向对象的程序设计

Java 是一种面向对象的程序设计语言，在该语言中：

- 每个数据项均封装在某个对象中。
- 每条可执行的语句均由某个对象来完成。
- 每个对象均是某个类的实例或是一个数组。
- 每个类均在一个单继承的层次结构中定义。

Java 中的单继承层次结构是一种树型结构，它的根是 object 类。Java 的面向对象的特点使得它特别适合于数据结构的设计和实现，为程序员提供了很好的开发平台。

1.6.2　变量和对象

一个 Java 程序由一个或多个文本文件组成，其中至少有一个文本文件包含了一个 public 类(惟一的一个)，该类中含有惟一的一个 main 方法，典型的用法是：

```
public static void main(string[]    args)
```

程序中的每个文件必须包含一个惟一的 public 类或 public 接口定义。对 X.java 的文件而言，X 就是该文件中定义为 public 的类名或接口名。文件编译后形成.class 的类文件。程序通过包含 main(string[])的 class 文件执行，程序既可以在命令方式下执行，又可以在集成开发环境中运行。

在程序设计语言中，数据的存取是通过变量实现的。在 Java 中，一个变量要么是对象的一个引用，要么是 8 种基本类型之一。如果变量是一个引用，那么它的值要么为空(null)，要么包含一个实例对象的地址。

每个对象属于一个特定的类，类定义了对象的特性和操作；一个变量是由说明其类型和初值的声明来定义的，对象由调用类构造器的 new 操作创建。

例 1：创建变量和对象。

```
public class ex
{public static void main(string[] args)
 {boolean flag=true;
  char   ch='a';
  short     m;
  int n=10;
  double xy;
  string str;
  string nns=null;
  string country = new string ("CHINA");
  system.out.println("falg="+flag);
  system.out.println("ch="+ch);
  system.out.println("n="+n);
  system.out.println("nns="+nns);
  system.out.println("country="+country);
  }
}
```

该例中定义了 8 个变量，前 5 个变量是基本类型变量，后 3 个变量是指向字符串的对象的引用。

Java 定义了 8 种基本类型：

- boolean　布尔型，其值为 false 或 true。
- char　字符型。
- byte　8 位整数，取值范围是 - 128~127。
- short　16 位整数，取值范围是 - 32768~32767。
- int　32 位整数。
- long　64 位整数。
- float　32 位浮点小数。
- double　64 位浮点小数。

1.6.3　流程控制

在条件分支中，Java 语言中有 if 语句、if…else 语句、switch 语句以及条件表达式运算符“？：”。

在循环中，Java 语句支持 while 语句、do…while 语句和 for 循环语句。

例如：分支和循环的使用。

```
public class examp1
 {
public static void main(string[] args)
    {
int n=(int )math.round(100*math.random());
    if (n>25)&&(n<75)
 system.out.println(n+"is between 25    and 75");
    else
 system.out.println(n+"is not between 25    and 75");
    switch ((int) n/20 )
      {
case 0:
system.out.println(n+"<20");
    case 1:
system.out.println(n+"<40");
    case 2:
system.out.println(n+"<60"); break;
    case 3:
system.out.println(n+"<80");break;
    default:
system.out.println(n+">=80");
    }
    while (n<=100)
    {
    {
system.out.println("n="+n);
    n++;
    }
    for (int j=1;j<=n;j++)
     system.out.println("n="+n);
  }
}
```

1.6.4　类和修饰符

一个类是一个抽象数据类型的实现，对象是通过对类进行实例化产生的，在类定义中指明了该类的对象所能保持的数据的种类和对象所能完成的操作，这些描述成为类的成员。

如 1.4 节中的 ADT 定义，一个类可以包含存储的数据集合，也可以包含在此数据集合上的一系列的基本操作。

对类、接口及其成员进行声明时，均可以带上修饰符，这些修饰符可以包含 public、protected、private、abstract、package、static 以及 final。

其中 public、protected、private 称做存取访问修饰符。因为它们将确定类或成员从哪些地方来进行访问。public 可以用来自任何地方的访问，protected 只能被本类或该类的子类的方法访问，private 则只能由该类自身访问。

当成员声明为 abstract 时，表明它是不完整的。意味着没有包括它的实现。应该说明的是：field 不能为 abstract。

一个 final 类是指不能有子类的类，而一个 final 的 field 表示它是一个常量。

一个 static 型的 field 成员是属于类本身的，而不是为该类的每个实例对象产生一个单独的备份。

1.7　算　　法

1.7.1　算法及其性质

如果问题是一个需要完成的任务，即对应一组输入就有一组相应的输出。只有问题被准确定义并完全理解后才能够研究问题的解决方法。例如，任何计算机程序只能使用可用的主存储器和磁盘空间，而且必须在合理的时间内完成运行。

从数学的角度讲，可以把问题看作函数。函数是输入(定义域，domain)和输出(值域，range)之间的一种映射关系。输入值称为函数的参数，不同输入可以产生不同的输出，但是给定的输入，每次必须有相同的输出。

算法是指解决问题的一种方法或者一个过程。一个问题可以用多种算法来解决，一个给定的算法解决一个特定的问题。

数据结构与算法之间存在着密切的关系。可以说不了解施加于数据上的算法需求就无法决定数据结构；反之算法的结构设计和选择又依赖于作为其基础的数据结构。即数据结构为算法提供了工具。算法是利用这些工具来实施解决问题的最优方案。因为在程序设计过程中相当多的时间花费在算法的构思上，一旦有了比较好的算法(解决问题的方法)，使用具体的程序设计语言实现不是很困难的事情。从这个角度来说，只有设计一个好的算法，

才有可能设计出一个好的程序。

简单地说，算法就是解决问题的方法，或算法是解决某个特定问题的一些指令的集合；或者说，由人们组织起来加以准备、加以实施的有限的基本步骤。流程图是图形化的算法，程序是用计算机语言描述的算法。

在计算机领域内，一个算法实质上是根据处理问题的需要，在数据的逻辑结构和存储结构的基础上施加的一种运算。因为数据的逻辑结构和存储结构不是惟一的，所以算法的描述可能不惟一。即使逻辑结构和存储结构相同，算法可能也不惟一。通过学习数据结构，可以使得程序设计者选择一种比较好的算法，当然好的算法是根据实际的硬件和软件环境进行衡量的。

一个完整的算法应该满足以下几条性质：

(1) 正确性。算法必须完成所期望的功能，得到的结果必须是正确的。

(2) 确定性。组成算法的指令必须是清晰的、无二义性。也就是说，算法的每一个步骤都必须准确定义。准确定义是指所描述的行为必须对人或机器而言是可读的、可执行的。每一步必须在有限的时间内执行完毕，同时必须是我们力所能及的，能够依赖于具体的工具来执行的工序。算法的指令必须具有可执行性，也就是说编写的算法必须是计算机能够执行，否则为无效算法。

(3) 有穷性。算法必须在有限的步骤内结束。如果一个算法由无限的步骤组成，该算法不可能有计算机程序实现。计算机只能解决有限问题，不能够解决无限问题。

(4) 输入。每个算法可以包含 $n(n\geq 0)$个数据的输入。

(5) 输出。算法至少有一个输出。

值得注意的是，一个算法可以没有输入(n=0)，但是至少应该有一个输出。如果算法没有了输出，也就没有结果，该算法就没有了实际意义。

程序可以看作是使用某种程序设计语言对一个算法的具体实现。所以就有人说程序=算法+数据结构。

1.7.2　算法描述的分析

1. 算法设计要求

设计一个好的算法通常应考虑达到以下目标：

(1) 正确性。算法设计应满足具体问题的需求。它至少应该包括对输入、输出及加工过程等的明确的无歧义性的描述。“正确”涵义通常包含程序无语法错误、程序能够得到正确的结果、程序对不合法的数据输入应有满足要求的结果。一般对程序测试时，以录入不合法数据得到满足要求的结果作为衡量程序是否合格的标准。

(2) 可读性。算法主要是为了阅读与交流，算法可读性好有助于对算法的理解。

(3) 健壮性。当输入非法的数据时，算法能够适当地作出反应或进行处理，不会产生莫名其妙的输出结果。

(4) 效率与低存储量需求。效率是指算法的执行时间，一般对问题的求解方法很多，执行时间短的算法效率高。存储量的需求是指算法执行过程所需要的最大的存储容量，存储空间越小，则算法越好。效率和低存储量两者通常情况下是矛盾的。也就是说，在编写算法时，有时为了追求较少的运行时间，可能会占用较多的存储空间；当追求较少的存储空间时，又可能带来运算时间增加的问题。因此在衡量算法好与坏时，应该综合考虑各个方面的因素，才能够得到较好的算法。时间和存储空间两者都与问题的规模有关。

2. 算法分析

算法是解决计算问题的工具。算法的好与坏直接影响解决问题的效率，为提高解决问题的效率，针对一个具体问题的解决方法，除了需要考虑对算法的具体描述外，还应具有衡量该算法好坏的方法。

一个算法的描述可以考虑各种具体的语言或类语言，一般情况下使用类语言的较多。所谓类语言是指使用高级语言的基本语法和指令，但并不能直接在机器上运行，如果需要运行，还需要按照高级语言的严格规定进行调试方可。例如：循环语句一般有 for 语句，不同的语言其格式可能不相同。

算法的分析主要是指判断算法的优劣，判断一个算法的好坏一般从两个方面考虑，即从时间角度和从空间角度上衡量算法。一般算法分析从时间角度考虑的比较多。当然判断一个算法的好与坏，不能以时间或空间衡量简单化，而是根据实际情况综合考虑。

度量一个程序的执行时间通常有两种方法：

(1) 事后统计方法。这种方法的缺点是，必须先运行依据算法编制的程序；所花时间的统计量依赖于计算机的软件、硬件等环境因素，容易掩盖算法本身的优劣。因此人们常用第 2 种方法。

(2) 事前分析估算的方法。一个用高级语言编写的程序在计算机上运行时所消耗的时间取决于下列因素。

- 算法选用何种策略
- 问题规模
- 书写程序的语言
- 编译产生的机器代码质量
- 机器执行指令的速度

一个算法的执行时间按照以上 5 点衡量是不合适的，因为在不同的计算机上运行同一个程序，其时间可能不同。但是如果撇开和计算机软件、硬件有关的因素，可以说算法效率依赖于问题的规模，或者说，它是问题规模的函数。

但在实践中，我们可以把两种方法结合起来使用。

一般地，我们将算法的求解问题的输入称为问题的规模，并用一个整数 n 表示。例如，矩阵乘积问题的规模是矩阵的阶数，而一个图论问题的规模则是图中的顶点个数或边的条数。

在算法的每个步骤中，可能有若干条语句，而频度就是指每条语句的执行次数。一个

算法的时间复杂度是指算法的时间耗费。简单地说，就是以一条基本语句的执行时间为基本单位，该算法所有语句中总的基本语句的执行次数就是该算法的时间耗费，它是该算法所求解的问题规模 n 的函数。当问题的规模 n 趋向无穷大时，我们把时间复杂度的数量阶称为算法的渐进时间复杂度。一般我们把渐进时间复杂度称为算法的时间复杂度，记作“O”(Order)，它有严格的数学定义：若 $T(n)$和 $f(n)$是定义在正整数集合上的两个函数，则 $T(n)=O(f(n))$表示存在正的常数 C 和 n_0，使得当 $n \geqslant n_0$ 时都满足 $0 \leqslant T(n) \leqslant C * f(n)$。

评价或分析一个算法的时间复杂度时，需要了解算法分析的一些主要概念。

例 1：下列程序。

```
…
for(int I=0;I<n;I++)   /*执行 n+1 次
for(int j=0;j<n;j++)   /*执行 n*(n+1)次
x=x+1;                 /*执行 n*n 次
…
```

以每条语句的执行次数计算，总的执行次数是 $2n^2+2n+1$ 次，当 n 趋向无穷大时，其执行次数和 n^2 是同一个数量阶，所以其时间复杂度记作 $O(n^2)$。

例 2：将程序改为以下格式。

```
…
for(int I=0;I<1000;I++)   /*执行 1001 次
for(int j=0;j<1000;j++)   /*执行 1000*1001 次
x=x+1;                    /*执行 1000*1000 次
…
```

此时 n 的值是一个常数，总的执行次数也是常数，按照 O 定义，和常数同阶，所以其时间复杂度应为 $O(1)$。

例 3：在一个有序的数组中查找关键字 K。

查找方法是从中间位置开始比较，该位置记为 mid，相对应的数据元素是 K_{mid}，比较结果有 3 种情况。

- $K_{mid}>K$：K 如果存在，肯定处于前半部分。
- $K_{mid}=K$：查找成功。
- $K_{mid}<K$：K 如果存在，肯定处于后半部分。

无论(1)、(3)哪种情况，都是忽略一半，缩小一半范围，重复这个过程，就能够找到指定的元素(或确定它不在数组中)，该过程至多重复 $\log_2^n$ 次。

下面是函数的过程：

```
static int binary(int K, int[] array, int left, int right)
{
  //返回 K 在数组中的下标(如果存在)，否则返回 - 1
  int l=left - 1;
```

```
int r=right+1;                    //l 和 r 是数组的上下界
while(l+1!=r) {                   //当 l 和 r 相遇时退出循环
int i=(l+r)/2;                    //取数组中的中间值比较
if(K<array[i]) r=i;               //如果小于，数据如果存在肯定在左半部分
if(K==array[i]) return i;         //已经找到 K
if(K>array[i]) l=i;               //K 可能在数组的右半部分
}
return  -1;                       //查找不成功
}
```

函数 binary 的功能是查找 K 的(惟一)位置，并返回该位置。若 K 不在数组中，则返回一个特定信息。还可以对此算法稍加改动，使之能够返回 K 在数组中第一次出现的位置(若数组元素允许有重复)，或者当 K 不在数组中时返回小于 K 的最大元素的位置。对比顺序检索和二分法检索，可以看到顺序检索法的平均和最差情况代价 $O(n)$将远远大于二分法的代价 $O(\log_2^n)$。孤立地看，二分法比顺序检索法效率高得多。但是，注意顺序检索法的时间代价总是相差不多的，无论数组中的元素是否按照顺序保存。相反，二分法检索要求元素必须按从低到高的顺序保存。根据二分法使用的环境，这个排序的要求可能会对时间代价产生损害，因为要保持数组的顺序性，在插入新元素时会增加时间代价。这里有一个权衡的问题：使用二分法检索比较容易，但是维持一个有序的数组比较费时，怎样权衡其利弊呢？只有在解决具体问题时才能知道是否利大于弊。

空间复杂度(Space Complexity)类似于时间复杂度，是指该算法所耗费的存储空间，也是问题规模 n 的函数。一般是指渐进空间复杂度。记作：

$$S(n)=O(f(n))$$

其中 n 是问题的规模(大小)。上机执行的程序除需要存储空间寄存本身所用的指令、常数、变量和输入数据等外，一般还需要数据进行操作的工作单元和实现计算所需要的信息的辅助空间。如果空间只是取决于问题本身，与算法无关，则只需要分析计算所需要的辅助空间，否则应考虑输入本身所需要的空间。

例 4：一个包含 n 个整数的一维数组空间代价是多少？

如果每个整数占有 c 个字节空间，则整个数组占用 cn 个字节空间，其空间复杂度是 $O(n)$。

在存储结构中，例如指针实际上是附加信息的开销，称为结构性开销。从理论上讲，这种结构性开销应该尽量小，而访问路径又应该尽可能多且有效。时间空间的权衡也正是研究数据结构的乐趣所在。

例 5：信息压缩和解压缩。

在信息的压缩或者加密过程中，节省了存储空间，但是在信息的处理过程中，又增加了时间上的开销。这样得到的程序空间代价小了，但时间的代价却大了。相反，许多程序

也可以预先存放部分结果或者对信息重组，以达到提高运行速度的目的，但代价是占用了较多的存储空间。通常情况下，这些时间空间上的变化都是通过常数因子来改变的。

算法分析时，除特别指明，均是按照最坏的情况分析。我们通常所说的算法的复杂度一般是算法的时间复杂度和空间复杂度的合称。

最好的时间空间改进来源于好的数据结构或算法，所以解决问题的步骤应该是：先调整算法，后调整代码。

思考和练习

1. 填空题类

(1) 数据结构是研究数据的_____和_____以及它们之间的相互关系，并对这种结构定义相应的_____，设计出相应的_____。

(2) 数据结构通常是指_____结构和_____结构两种，通常是指_____结构。

(3) 数据结构按结点间的关系，可分为 3 种逻辑结构，它们分别是____、____和____。

(4) 选择合适的存储结构，通常考虑的指标有_____和_____两个因素。

(5) 数据结构的内存存储方式主要有_____和_____两种。

(6) 线性结构反映结点间的关系是_____对_____的，树型结构反映结点的关系是_____对_____的，网状关系反映结点的关系是_____对_____的。

2. 简答题类

(1) 叙述算法的定义及其重要特性。

(2) 分析下列程序，并用 O 表示时间的复杂度。

①
```
I=1; k=0
while (I<n)
{k=k+10*I;
I++; }
```

②
```
for (I=1; I<=n; I++)
for (j=1; j<=I; j++)
for (k=1; k<=j; k++)
x=x+1;
```

③
```
I=0; k=0; n=100;
while (I<=n)
{k=k+10*I;
I=I+1}
```

④ 假设数组 A 元素是从 0 到 $n-1$ 的任意一个排列。

```
Sum=0;
If   (even(n))
      For(i=0;i<n;i++)
            Sum++;
Else
      Sum=sum+n;
```

(3) 简述线性结构、层次结构、网状结构的不同点。

(4) 简述算法复杂度评价方法。

(5) 设有两个算法在同一机器上运行，其执行时间分别为 $100n^2$ 和 $2n$，要使前者快于后者，n 至少要多大？

(6) 算法的时间复杂度仅与问题的规模相关吗？

(7) 常用的存储表示方法有哪几种？

(8) 从以前用过的程序中找出一个慢得无法接受的程序，找出使程序速度慢的个别操作和使程序执行得足够快的其他基本操作。

(9) 大多数程序设计语言有内建的整数数据类型：在正常情况下这种表示方法有固定的长度(表示一个整数所用的位数)，因此限制了整型变量值的大小。给出一种无长度限制(除计算机可用内存的限制之外)的整数表示方法，这样存储的整型变量大小就不受限制了。试用这种表示方法简要地说明如何实现加、乘、指数操作。

(10) 为字符串定义一个 ADT，要求包含字符串的典型操作，每一个操作定义为一个函数。每一个函数由它的输入、输出来定义。

(11) 为一个整数线性表定义一个 ADT。它包含线性表常用的操作(一个函数对应一个操作)，每一个函数由它的输入、输出定义。

(12) 为一个整数集合定义一个 ADT(注意集合中无重复元素)。它包含整数集合的常见运算，每一个运算对应一个函数，每一个函数由它的输入、输出定义。

(13) 为二维整数数组定义一个 ADT，详细而准确地说明可以在此数组上完成的操作。然后，试着将它用于一个 1 000 行、1 000 列的数组，其中非零元素远少于 10 000 个，为这个数组设计两种不同的实现方法，使它们比使用标准的二维数组所需要的 100 万(1 000*1 000)个位置的实现方法节省空间。

(14) 每一个问题都有一个算法吗？

(15) 设计一个在家用电脑上运行的拼写检查程序，它能够快速处理少于 20 页的文献。假设这个程序有一个大约 20 000 字的以 ASCII 码形式存储的字典。这个字典必须实现的原语操作是什么?每一个操作的合理时间限制是多少？

(16) 设想你被雇用去设计一个包含美国城市和乡村信息的数据库服务系统，有成千个城市和乡村。此数据库程序应当允许用户按照名字检索某一个地方的信息，用户根据位置或人口等属性的某一个特定值或某一个范围内的值来查找满足条件的所有地方。按照用户应该看到的操作，描述系统的基本功能，并列出每个操作的时间限制。

(17) 尽可能地写出你所能想到的对 1 000 个数进行排序的方法。其中哪一种(些)最好？

(18) 一个图由顶点集合和边集合组成，每条边连接两个顶点，任意一对顶点间只能连接一条边。至少给出两种不同的方法，在计算机中表示由图的顶点和边定义所确定的连接关系，你的表示方法要能够用来确定给定的一对顶点间是否有边相连。

(19) 假设有一组记录，按照每个记录都包含的一些关键码字段排序。给出两种不同的方法检索有特定关键码值的记录，你认为哪种更好？为什么？

(20) 怎样比较两种对一组整数进行排序的算法？特别地，①作为比较两种排序算法的基础，用什么代价度量比较合适？②在这些代价度量方式下，用什么测试方法来判断这两种算法的性能？

(21) 按照增长率从低到高的顺序排列以下表达式：

$4n^2$，$\log_3 n$，$3n$，20^n，2，$\log_2 n$，$n^{2/3}$

(22) 画出下列函数图，并说出当 n 取什么值时，每个表达式的效率最高。

$4n^2$，$\log_3 n$，$3n$，20^n，2，$\log_2 n$，$n^{2/3}$

(23) ① 假设某一算法的时间开销为 $T(n)=3*2n$，对于输入规模 n，在某台计算机上实现并完成该算法的时间为 T 秒。现在另有一台计算机，运行速度为第一台的 64 倍，那么 T 秒内新机器上能完成输入规模为多大的问题？

② 假设又有一种算法，$T(n)=n^2$，其余条件不变，那么新机器 T 秒内能完成的输入规模是多少？

③ 若算法 $T(n)=8n$，其余条件不变，那么在新机器上 T 秒内能够处理多少输入数据？

(24) 硬件厂商 XYZ 公司宣称他们最新研制的微处理器运行速度为其竞争对手 Prunes 公司同类产品的 100 倍。若 Prunes 公司的计算机能在 1 小时内完成输入规模为 n 的某程序，对算法增长率分别为 n、n^2、n^3、$2n$ 的程序，分别计算 XYZ 公司的计算机 1 小时内能完成的输入规模。

(25) 根据 O 的定义，写出下列表达式的最好和最坏的情况。

① $c_1 n$

② $c_2 n^3+c_3$

③ $c_4 n\log n+c_5 n$

④ $c_6 2^n+c_7 n^6$

第2章　线　性　表

线性表是我们日常工作中最简单也是最常用的一种数据结构，它的最基本的特点是每个数据元素最多只能有一个直接前趋，每个数据元素最多只能有一个直接后继；只有第一个数据元素没有直接前趋，而最后一个数据元素没有直接后继。

本章的学习目标：

- 线性表的数据结构，线性表最显著的特点；
- 线性表的两种主要的存储结构在存储时的地址分配；
- 顺序表存储的数据元素的地址的计算和在顺序存储结构下的基本运算；
- 链式存储结构下，线性表运算的优缺点；
- 链式存储结构的基本运算的实现。

2.1　线性表类型的定义

线性表是最常用、最简单的一种数据结构，简言之，线性表是 n 个数据元素的有限序列。其一般描述为：

$$A=(a_1, a_2, \cdots, a_n)$$

其中，A 称为线性表的名称，每个 $a_i(n\geqslant i\geqslant 1)$称为线性表的数据元素，具体 n 的值含义则称为线性表中包含有数据元素的个数，也称为线性表的长度；当 n 的值等于 0 时，表示该线性表是空表。每个数据元素的含义在不同情况下各不相同，它们可能是一个字母、一个数字，也可以是一条记录等。一般情况下，在线性表中每个 a_i 描述的是一组相同属性的数据。

线性表的离散定义是：$B=<A, R>$，其中 A 包含 n 个结点$(a_1, a_2, \cdots\cdots, a_n)$，$R$ 中只包含一个关系，即线性关系。$R=\{(a_{i-1}, a_i)|\ i=1, 2, \cdots\cdots, n\}$，线性表中包含的数据元素个数为线性表的长度。

一个数据元素通常包含多个数据项，此时每个数据元素称为记录，含有大量的记录的线性表称为文件。

例如 26 个英文字母表$(A, B, \cdots\cdots, Z)$是一个线性表，表中的数据元素是字母。

在稍微复杂的线性表中，一个数据元素可以由若干个数据项组成。

例如，一个学生的档案管理表如图 2-1 所示，每个数据元素相当于表中的一行，包含有姓名、学号、出生年月、性别、专业、籍贯等 6 个数据项。

学号	姓名	出生年月	性别	专业	籍贯
890001	陈红	1980.2	女	计算机	山东省
890002	王小康	1981.3	男	计算机	北京市
890004	张三	1982.6	男	计算机	湖南省
890005	赵屋	1980.5	男	计算机	湖北省
……	……	……	……	……	……

图 2-1 学生档案表

从上例中可以看出每个数据元素具有相同的特性，相邻的数据元素之间存在着序偶关系，即每个数据元素最多只能有一个直接前趋元素，每个数据元素最多只能有一个直接后继元素；只有第一个数据元素没有直接前趋元素，而最后一个数据元素没有直接后继元素，线性表中的数据元素个数(学生人数)是该线性表的长度。

线性表是一个比较灵活的数据结构，它的长度根据需要增长或缩短，也可以对线性表的数据元素进行不同的操作(如访问数据元素，插入、删除数据元素等)。

一般对线性表的操作可以包含以下几种：

- Pubilc linklist() 建立一个空的线性表。
- Pubilc linklist(collection c) 将 collection c 中的数据依次建立一个线性表。
- Pubilc object getfirst() 返回线性表的第一个元素。
- Pubilc object getlast() 返回线性表的最后一个元素。
- Pubilc object removefirst() 删除线性表的第一个元素，并将该值返回。
- Pubilc object removelast() 删除线性表的最后一个元素，并将该值返回。
- Pubilc void addfirst(object o) 将 object o 插入在链表的开头位置。
- Pubilc void addlast(object o) 将 object o 插入在链表的末尾位置。
- Public boolean contains(object o) 检查 object o 是否在链表中，如果存在返回 true，反之返回 false。
- Public int size() 返回线性表的元素个数。
- Public boolean add(object o) 将 object o 插入到链表的末尾，并返回 true。
- Public boolean remove(object o) 将 object o 在链表中第一次出现的元素删除，成功返回 true，否则返回 false。
- Public addall(collection c) 将 collection c 中的数据依次插入到链表的末尾。
- Public addall(int index, collection c) 将 collection c 中的数据依次插入到链表的 index 位置，并将 index 位置之后的元素依次插入在 collection c 之后，连接成一个完整的线性表。
- Public void clear() 删除线性表中的所有元素。
- Public object get(int index) 返回线性表的 index 位置的元素。
- Public object set(int index, object element) 以 object element 取代线性表 index 位置的元素。

- Pubilc void add(int index, object element)　将 object element 插入到线性表 index 位置之后。
- Pubilc object remove(int index)　删除线性表中的 index 位置的元素。
- Public int indexof(object o)　返回 object o 在线性表第一次出现的位置，若不存在返回 - 1。
- Public int lastindexof(object o)　返回 object o 在线性表最后一次出现的位置，若不存在返回 - 1。
- Public listiterator listiterator(int index)　返回线性表 index 位置开始的元素内容。

使用抽象数据类型(ADT)定义线性表如下：

ADT list{
数据对象:D={a_i | a_i∈元素集合, i=1, 2, ……, n, n≥0}
数据关系:R= {<a_{i-1}, a_i> | a_{i-1}, a_i∈元素集合, i=1, 2, ……, n}
基本操作:
{将以上对线性表的操作搬下来，每个函数注明输入输出}此处可以写得详细一些。
}ADT list

注意：

以上只是线性表的基本运算，对于线性表其他的运算，可以通过基本运算的组合来实现。例如将两个或两个以上的线性表合并成一个线性表，把一个线性表拆分成多个线性表等。

2.2　线性表的顺序表示和实现

线性表的存储结构分为顺序存储和非顺序存储。其中顺序存储也称为向量存储或一维数组存储。

1. 顺序表

线性表的顺序存储，也称为向量存储，又可以说是一维数组存储。线性表中结点存放的物理顺序与逻辑顺序完全一致，它叫向量存储(一般指一维数组存储)，与此同时对应 $A=(a_1, a_2, \cdots, a_n)$线性表而言，线性表的存储结构如图 2-2 所示。

a_1	a_2	……	a_n	……

图 2-2　线性表的存储结构图

实际上，数据的存储逻辑位置由数组的下标决定。所以相邻的元素之间地址的计算公式为(假设每个数据元素占有 c 个存储单元)：

$$LOC(a_{i+1})=LOC(a_i)+ c$$

对线性表的所有数据元素，假设已知第一个数据元素 a_1 的地址为 $d1$，每个结点占有 c

个存储单元，则第 i 个数据元素 a_i 的地址为：

$$di=d1+(i-1)*c$$

线性表的第一个数据元素的位置通常称做起始位置或基地址。

线性表的这种机内表示称做线性表的顺序存储结构或顺序映像(Sequential Mapping)，使用这种存储结构存储的线性表又称做顺序表。其特点是，表中相邻的元素之间具有相邻的存储位置。

值得注意的是，在使用一维数组时，数组的下标起始位置根据给定的问题确定，或者根据实际的高级语言的规定确定。

顺序分配的线性表可以直接使用一维数组描述为：

```
type arraylist[];   //type 的类型根据实际需要确定//
```

通常，用在数组的元素个数不是很多且可以对数组元素“枚举”的情况下，也可以使用符合类型数组的动态进行动态定义。

```
type arrayname[];
```

该代码只是对应用数组的声明，还没有对该数组分配空间，因此不能访问数组。只有对数组进行初始化并申请内存资源后，才能够对数组中元素进行使用和访问。

```
arrayname= new type[arraysize];
```

其作用是给名称为 arrayname 的数组分配 arraysize 个类型为 type 大小的空间。其中 arraysize 表示数组的长度，它可以是整型的常量和变量。如果 arraysize 是常量，则分配固定大小的空间；如果它是变量，则表示根据参数动态分配数组的空间。

数组的初始化也可以使用静态初始化，也就是在数组定义的同时对数组元素赋值。例如：

```
int arrayname[]={0,1,2,3,4};
```

在引用数组的元素时，和其他高级语言的用法相同，可以直接使用数组的下标表示一个数据元素的值和位置，例如：arrayname[index]，其中 arrayname[index]表示该数据元素的值，而 index 表示数组在 arrayname 中的下标(位置)。下标的范围从 0 开始一直到数组的最大长度减 1，数据类型则可以是整型常量、变量和表达式。在 Java 中可以直接使用数组的属性 length 获得数组的长度。

2. 顺序表基本运算的实现

线性表顺序存储的结构容易实现线性表的某些操作，如随机存取第 i 个数据元素等，但是在插入或删除数据元素时则比较繁琐，所以顺序存储结构比较适合存取数据元素。应该注意 Java 的数组下标从 0 开始。下面考虑线性表顺序存储的插入、删除和排序的实现方法。

例 1：在线性表的第 $i-1$ 个数据元素和第 i 个数据元素之间插入一个新的数据元素，使线性表的长度增加一个元素的空间。

算法分析：用顺序表作为线性表的存储结构时，必须保证数据存储的连续性，必须从第 i 个元素到第 narray.length - 1(因为下标从 0 开始)个元素向后平移，空出一个存储单元后，插入该元素(如图 2-3 所示)。

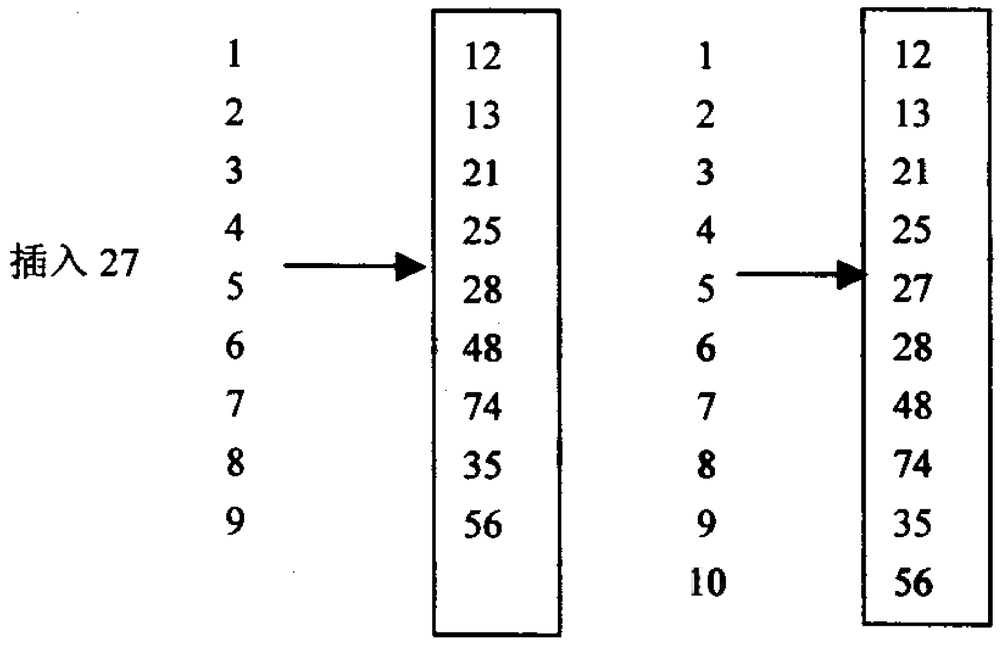


图 2-3　顺序存储插入前后状况图

注意：

对数据元素进行平行向后移时，遵循存储单元存储的后入为主的原则，否则将造成数据丢失。

对于算法实现时，主要是一个平移的过程，应该从后向前移动，不应该反向移动；同时应该判断 i 的取值是否合理。另外，还应该考虑当前的数据个数应该最多有 narray.length - 1 个，否则就要溢出了。在程序段的适当处应加注释。下文同，不再一一写明。

```
Public class arrayinsert
{ public static void main(string args[])
   { int narray[];
     if (i>narray.length-1&&i<0)
       system.out.println("i not exist")
     else
       {for (int k=narray.length-1; k<=i; k--)
        narray[k+1]=narray[k]
        narray[i]=item;
        }
     narray.length=narray.length+1;}
  }
```

注意：

插入结束后要修改线性表的长度。以上算法的时间主要花费在结点后移语句 for 循环上。该语句最坏的情况下，移动次数是 narray.length，最好的情况下是 0，因为整个循环的

次数是 narray.length-i+1，所以时间不仅与规模长度有关，而且与插入的位置有关。其时间复杂度是 $O(n)$。

例 2：顺序存储结构中，删除线性表的第 i 个数据元素，使线性表的长度减少一个元素的空间。作分析的时候可以给出直观的图来表示，以便于理解。

算法分析：用顺序表作为线性表的存储结构时，必须保证数据存储的连续性，必须从第 narray.length - 1(因为下标从 0 开始)个元素到 i 个元素向前平移，直接删除一个数据元素。

注意平移时，遵循存储单元存储的后入为主的原则，需要从前向后移动，否则将造成数据丢失。

对于算法实现时，主要是一个平移的过程，应该从前向后移动；同时应该判断 i 的取值是否合理。另外，还应该考虑当前的数据个数是否为 0，否则就要溢出了。

```
Public class arraydelete
{ public static void main(string args[])
   { int narray[];
     if (i>narray.length&&i<0)
       system.out.println("i not exist")
     else if (narray.length==0)
              system.out.println("overflow")
            else
     {for (int k=i ; k<narray.length-1; k++)
     narray[k]=narray[k+1]
     }
   narray.length=narray.length-1;}
}
```

例 3：显示顺序存储的线性表的所有元素。

算法分析：在该例中，从下标为 0 的位置开始打印，线性表的最后一个数据元素可以通过 length 获得，只要用循环语句实现即可。只不过需要判断当前线性表是否为空，如果为空，显示一个特殊的符号即可。

```
Public void print()
{
 if (isempty())
    system.out.print("()");
 else
    {
      system.out.print("(");
     for(int i=0;i<=narray.length-1;i++)
      system.out.print(narray[i]+ " ");
      system.out.print(")");
}
}
```

从以上算法例 1 和例 2 可见，当在顺序存储结构的线性表中某个位置上插入或删除一个数据元素时，其时间主要耗费在移动元素上(换句话说，移动元素的操作为预估算法时间复杂度的基本操作)，而移动元素的个数取决于插入或删除元素的位置。

假设 p 是在第 i 个元素之前插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需移动元素的平均次数为：

$$E1=\sum p(n-i+1)$$

假设 q 是删除第 i 个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素的平均次数为：

$$E2=\sum q(n-i)$$

假定在线性表的任何位置上插入或删除元素都是等概率的，即：

$$p=1/(n+1) \qquad q=1/n;$$

则 $E1$ 和 $E2$ 可分别简化为以下式子：

$$E1=n/2$$
$$E2=(n-1)/2$$

由以上的式子可见，在顺序存储结构的线性表中插入或删除一个数据元素，平均约移动表中一半元素。若表长为 n，则插入和删除算法的时间复杂度为 $O(n)$。

由此可以讨论在顺序存储结构下其他基本运算的实现方法和时间复杂度。顺序表的“求表长”和取第 i 个数据元素的时间复杂度均为 $O(1)$，因为可以直接求出线性表的长度，顺序存储下可以实现随机存取，可以直接取得数据元素，而不需要移动元素。

2.3　线性表的链式存储结构

线性表的顺序存储结构的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，因此随机存取元素时比较简单，但是这个特点也使得在插入和删除元素时，造成大量的数据元素移动，同时如果使用静态分配存储单元，还要预先占用连续的存储空间，可能造成空间的浪费或空间的溢出。如果采用链式存储，就不要求逻辑上相邻的数据元素在物理位置上也相邻，因此它没有顺序存储结构所具有的缺点，但同时也失去了可随机存取的优点。

2.3.1　单向链表

任意存储单元存储线性表的数据元素，对于链式存储线性表时，其特点形式如图 2-4 所示。

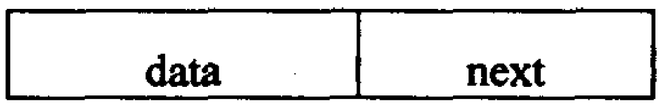


图 2-4 链式存储线性表逻辑结构图

其中 data 是数据域，存放数据元素的值；next 是指针域，存放相邻的下一个结点的地址，单向链表是指结点中的指针域只有一个沿着同一个方向表示的链式存储结构。

因为结点是一个独立的对象，所以能够实现独立的结点类。以下是一个结点类的定义：

```
class link {
   private link next;
   private object data;
   link(object it , link nextval)
   { data=it ;next=nextval;}
   link(nextval){next=nextval;}
   link next(){return next;}
   link setnext(link nextval){return next=nextval;}
   object data(){return data;}
   object setdata(object it){return data=it;}
}
```

link 类比较简单，其构造函数有两种形式，一个函数有初始化元素的值，而另一个没有。其他函数帮助用户访问两个(私有)数据成员。使用这个类的用户可以取得并设置 next 和 data 域的值。由于这些设置(set)函数可以用来控制值的改变，只接受合理的赋值即可。link 类实际上是链表的实现，而不是线性表类的公共接口。

对于链式分配线性表，整个链表的存取必须是从头指针开始，头指针指示链表中第一个结点的存储位置。同时由于最后一个数据元素，没有直接后继，则线性链表中最后一个结点的指针为“空”(null)。

在使用单链表结点时，必须完成 3 步(如图 2-5 所示)：

(1) 创建一个新结点；

(2) 为该结点赋一个新值，将当前元素的 next 域赋给新结点的 next 域；

(3) 当前结点的前趋的 next 域要指向新插入的结点。

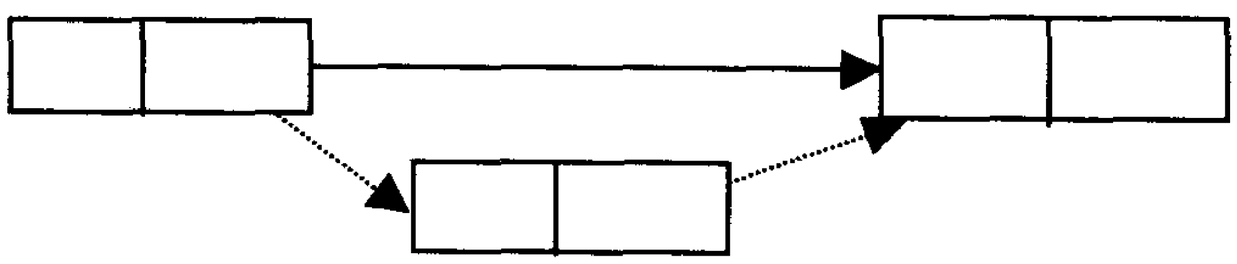


图 2-5 单向链表的插入图

如果使用 curr 指针已经指向当前元素的前趋结点，以下命令完成了以上的 3 步操作：

```
cur.setnext(new link(it,cur.next()));
```

运算符 new 创建了链表的一个新结点，同时使得新结点的 next 域指向当前结点。而函

数 curr.setnext()则使当前结点的前趋结点的 next 域指向新插入的结点，完成了新结点的申请、插入过程。

如果可利用空间表使用 link 类实现，申请一个结点的算法为：

```
static link freel=null;
static link get(object it ,link nextval)
{
if (freel==null)
   return new link(it,nextval);
link temp=freel;
freel=freel.next();
temo.setdata(it);
temp.setnext(nextval);
return temp;
}
```

在使用完结点后(一般指被删除的结点)，删除过程是将当前结点的直接前趋结点的 next 域指向被删除结点的直接后继结点(如图 2-6 所示)。

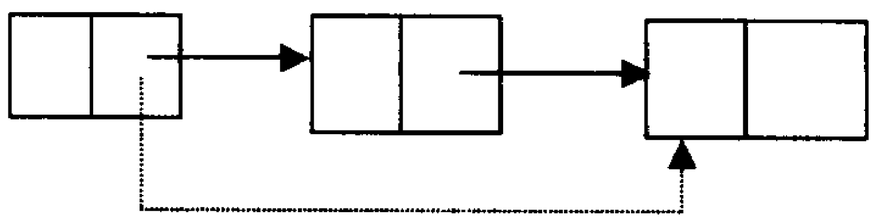


图 2-6　单向链表的删除逻辑图

如果 curr 表示当前结点的直接前趋结点，对应的删除命令是：

```
curr.setnext(curr.next().next());
```

该命令设置 curr 结点的 next 域为当前结点的的直接后继。

释放被删除结点的算法一般为：

```
void release()
{
   data=null;
   next=freel;
   freel=this;
}
```

删除结点只是将结点从链表中删除，该结点仍然存在，所以不能“丢失”被删除的结点的内存，需要将结点保留以便返回给空闲存储器。为便于将删除的结点返回，需要将被删除的结点指针赋值给临时的指针。

在可利用空间表中，可以使用以上的 get 和 release 函数完成申请和释放结点。但是对于可利用空间表的管理，实际上是对链表(假设可利用空间表是以链表的形式存储)的插入

和删除结点的过程。

调用 link 类的 get()形式是：

```
curr.setnext(link.get(it,curr.next()));  //得到一个结点 curr
```

调用 link 类的 release()形式是：

```
link tempptr=curr.next();
tempptr.release();  //释放 tempptr
```

2.3.2 单链表的基本运算

1. 建立链表

假设给定线性表中结点的数据类型是字符类型，逐个输入这些字符，每输入一个字符，将该字符赋值给新结点，输入字符时以换行符为输入结束标志符。

因为单向链表的长度不固定，所以应采用动态建立单向链表的方法。动态建立单向链表的常用方法有如下两种。

(1) 尾插入法

该方法是将新结点插到当前链表的表尾上，为此必须增加一个尾指针 tail 的开销，使其始终指向当前链表的尾结点；或者增加一个循环用来查找链表的末尾结点，然后将新结点插入到链表的末尾。此方法的优点是，在固定头 head 指针后，该指针就不能再变了。不会因为不断修改头指针，造成头指针的丢失。

实际上动态建立链表的过程，就是不断插入新结点的过程。尾插入结点的逻辑图如图 2-7 所示。

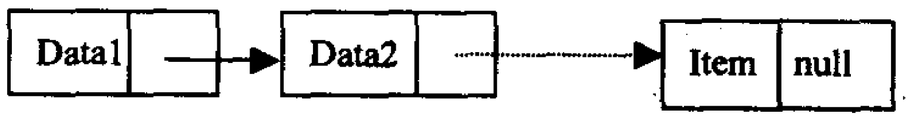


图 2-7 插入结点示意图

需要按照虚线的方式修改指针。

算法分析：动态建立链表的步骤应该是，第一个生成的结点是开始结点，将开始结点插入到空表中，是在当前链表的第一个位置上插入，该位置上的插入操作和链表中其他位置上的插入操作处理是不一样的(实际上对其他操作亦可能如此)，因为开始结点的位置是存放在头指针 head(指针变量)中，而其余结点的位置是在其前趋结点的指针域 next 中。因此必须对第一个位置上的插入操作做特殊处理。其他情况在 else 语句，直接插入即可。

建立链表需要读入若干数据(假设数据存放在 string name[]中)，若读入的第一个字符就是结束标志符，则链表 head 是空表，尾指针 tail 亦为空，结点 tail 不存在；否则链表 head 非空，最后一个尾结点 tail 是终端结点，应将其指针域置空。

算法如下：

```
class llist implements list
{ private link head;
   private link tail;
   protected link curr;
  llist(int sz){setup();}
  llist(){setup();}
  private void setup()//初始化
  { tail=head=curr=new link(null);
  }
public static void main(string args[])
{
  int dataname=0;
string dataname;
while (true)
  {dataname++;
  system.out.print("please input the data name:");
  dataname=console.readline();
  if (dataname.equals("0"))
  break;
newlist.insert(datanname)}
  }
public void clear()
{ head.setnext(null);
  curr=tail=head;//重新初始化
}
  public void insert(object it)//插入一个元素
{
assert.notnull(curr,"no current element");
curr.setnext(new link(it,curr.next()));
if (tail==curr)
  tail=curr.next();
}
  public void append(object it)//末尾添加一个元素
  {tail.setnext(new link(it,null));
  tail=tail.next;//末尾指针后移
  }
```

(2) 头插入法

算法分析：如果在链表的开始结点之前附加一个结点，并称它为头结点，那么会带来以下两个优点。

第一，由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其他位置上的操作一致，无须进行特殊处理。

第二，无论链表是否为空，其头指针是指向头结点的非空指针(空表中头结点的指针域空)，因此空表和非空表的处理也就统一了。在算法上，也就统一处理了(如图 2-8 所示)。

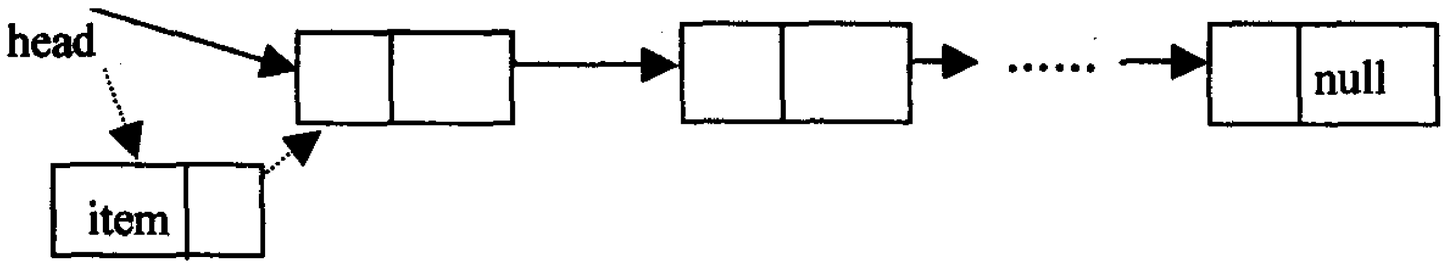


图 2-8 头插入法逻辑表示图

建立链表基本和尾插入结点方法类似。

```
public void insert(object it)//插入一个元素
{
assert.notnull(curr,"no current element");
curr.setnext(new link(it,head));
if (head!=curr)
 head=curr; //头指针前移
}
```

头插入法建立单向链表，相对尾插入法建立链表，不需要增加 tail 的开销，相对节省空间，但需要不断修改头指针。

2. 查找运算

(1) 按序号查找:

算法分析：在链表中，即使知道被访问结点的序号 i，也不能像顺序表中那样直接按序号 i 访问结点，而只能从链表的头指针出发，顺着链域 next 逐个结点往下搜索，直至搜索到第 i 个结点为止(一般采用计数器的方式)。链表不是随机存取结构，只能顺序存取。

查找之前首先要做到从头(head)开始，然后再逐个向后查找，查找过程中，每向后移动一次，计数器增加 1，直到找到第 i 个结点(查找成功)或找完整个链表没有第 i 个结点(查找失败)。

```
public void nexti()
{int j=0;
 curr=head;
 while ((j!=i)&&(curr!=null))
{ j++;
 curr=curr.next;
}
if (curr==null)
 return (0);
else
 return j;
}
```

仅当 1≤*i*≤*n*(*n* 为链表长度)时，*i* 值是合法的。但有时需要找头结点的位置，故我们把头结点看做是第 0 个结点，我们从头结点开始顺着链域扫描，用指针 curr 指向当前扫描到的结点(原因是头结点指针不能变)，用 *j* 作计数器，累计当前扫描的结点数，直至找到 *i* 结点。

curr 的初值指向头结点，*j* 的初值为 0，当 *p* 扫描下一个结点时，计数器 *j* 相应地加 1。当 *j*=*i* 时，指针 curr 所指的结点就是要找的第 *i* 个结点。

在算法中，while 语句的终止条件是搜索到表尾或者满足 *i*，当搜索到表尾，*j*!=*i* 时，表示 *i* 的取值不合理，超出了线性表的长度范围。

(2) 按数值查找

算法分析：查找结点有时可以按数值查找，按数值查找是在链表中，查找是否有结点值等于给定值 key 的结点，若有的话，则返回首次找到的其值为 key 的结点的存储位置；否则返回 null。查找过程从开始结点出发，顺着链域逐个将结点的值和给定值 key 作比较，有两种情况，curr.val=key(查找成功)；查到 curr=null 也没有出现 curr.val=key 的条件(查找失败)。

```
public void nextval()
{ curr=head;
  while ((curr.val!=key)&&(curr!=null))
{ curr=curr.next;
}
if (curr==null)
  return (null);
else
  return (curr);
}
```

3. 求链表长度

算法分析：求链表长度基本同按序号查找，从头结点开始顺着链域扫描，用指针 curr 指向当前扫描到的结点(原因是头结点指针不能变)，用 len 作计数器，累计当前扫描的结点数，直至 curr=null，返回长度 len 就可以了。

```
public int length()
{ int len=0;
  curr=head;
  while (curr!=null)
{curr=curr.next;
len++; }
return len;
}
```

4. 删除结点

算法分析：删除结点是将表中的某个结点从表中删除，实际上还是利用查找算法，找到满足条件的将要删除的结点后(注意删除过程中，使用的指针是被删除结点的直接前趋结点的指针)直接删除即可。

删除结点可按如图 2-9 虚线所指修改指针。

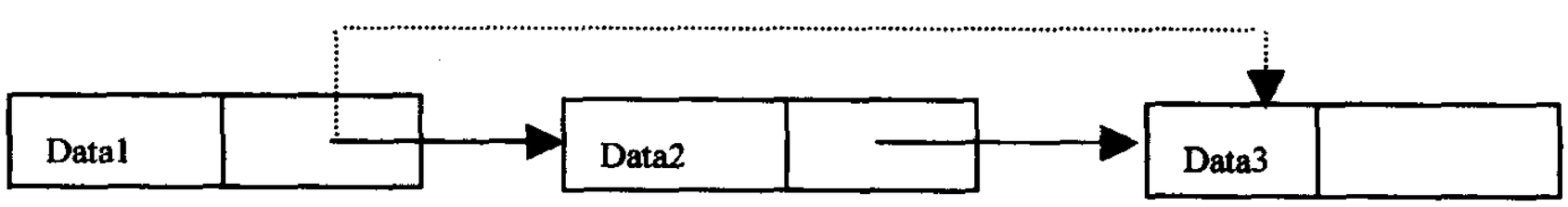


图 2-9 删除结点示意图

假设当前结点已经找到，curr 表示当前结点的直接前趋结点。

```
public object removet()
{ if (!((curr!=null)&&(curr.next()!=null)) return null;
  object it=curr.next().data();
if(tail==curr.next())    tail=curr;
curr.setnext(curr.next().next());     //删除结点
return it;
}
```

5. 打印链表的所有元素

算法分析：打印链表的所有结点的数值，与求链表的长度的方法基本类同，只是在找到每个结点的后面增加一条打印命令，去掉计数命令，在此方法中需要特别处理的是链表为空时的情况。

```
public void prin()
{curr=head;
  if (head.next.next()==null)
    system.out.print("this is a empty list");
else
  {while (curr.next!=null)
{system.out.print(curr.next().data());
 curr=curr.next;}
}}
```

从以上的例子中可以发现，在整个单向链表的所有操作中，只要做到单向链表的初始化后，剩下比较重要的算法是对链表的插入、删除和查询操作。只要比较灵活地掌握插入、删除和查询 3 个基本的操作，其他大部分操作可以利用以上的 3 种基本操作组合实现。

值得注意的是，在单链表中，因为指针是单一方向，结点的查找只能从前向后查找，不能反向查找，所以在插入、删除结点时，特别是在某个结点之前插入，或者删除某个结

点时，需要利用结点的前趋结点的指针，在查找结点时，需要保留结点的直接前趋结点位置。也因为在单链表中，结点的查找只能从前向后查找，不能反向查找，所以在单向链表中，头结点非常重要，不能丢失。

2.3.3　循环链表

循环链表又称为循环线性链表，其存储结构基本同单向链表。它是在单向链表的基础上加以改进形成的，可以解决单向链表中单方向查找的缺点。因为单向链表只能沿着一个方向，不能反向查找，并且最后一个结点指针域的值是 null，为解决单向链表的缺点，可以利用末尾结点的空指针完成前向查找。将单链表的末尾结点的指针域的 null 变为指向第一个结点，逻辑上形成一个环型，该存储结构称之为单向循环链表。如图 2-10 所示。

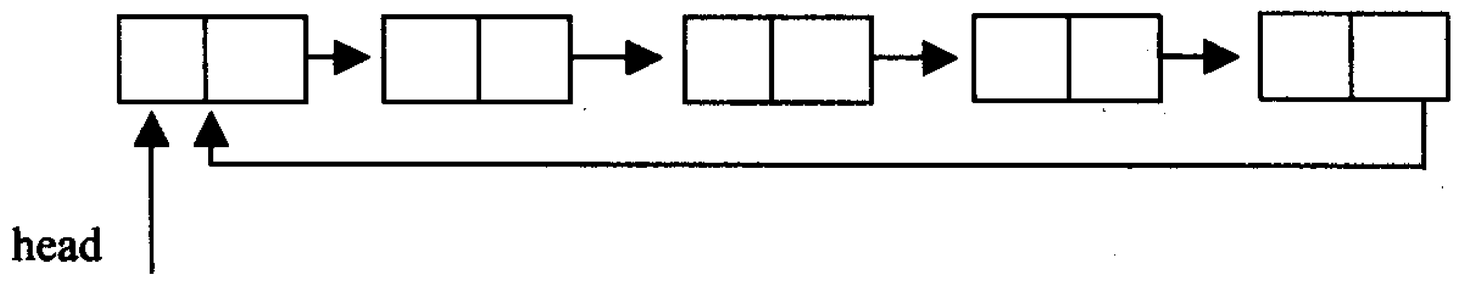


图 2-10　循环链表的逻辑结构图

它相对单链表而言，其优点是在不增加任何空间的情况下，能够已知任意结点的地址，可以找到链表中的所有结点(环向查找)。

当然在查找某个结点的前趋结点时，需要增加时间开销完成，查找的时间复杂度是 $O(n)$。

循环线性链表中已知链表中任何结点，可以找到链表中的所有结点，我们一般还是习惯把头结点作为已知条件，但是如果已知条件是头结点，将在以下的插入或删除结点时造成不方便：

- 删除末尾结点
- 在第一个结点前插入新结点

在第 1 种情况下，虽然能够完成删除，但是，需要我们从头结点开始逐个查找结点直到找到最后一个结点的直接前趋结点，然后才能够删除，整个算法的时间复杂度是 $O(n)$。

在第 2 种情况下，虽然能够完成插入，但是，需要我们从头结点开始逐个查找结点直到找到最后一个结点，然后才能够插入，因为我们需要修改最后一个结点的指针域，整个算法的时间复杂度是 $O(n)$。

以上两种情况造成无谓的时间开销，为解决这个问题，通常在循环链表以末尾结点指针为已知条件，这样以上的两种情况都可以直接完成，因为已知末尾结点可以直接找到头结点，此时的时间复杂度为 $O(1)$，这样在不增加任何开销的情况下，减少了时间的开销。

空的循环线性链表根据定义可以与单向链表相同，也可以不相同。判断循环链表的末尾结点条件也就不同于单向链表，不同之处在于单向链表是判别最后结点的指针域是否为空，而循环线性链表末尾结点的判定条件是其指针域的值指向头结点。

循环链表的插入、删除运算基本同单向链表，只是查找时判别条件不同而已。但是这种循环链表实现各种运算时的危险之处在于：链表没有明显的尾端，可能使算法进入死循环，所以判断条件应该用 curr.next()!=head 替换单向链表的 curr.next()!=null 完成遍历所有结点。

2.3.4 双链表

单循环链表中，虽然从任一已知结点出发能找到其直接前趋结点，但时间耗费是 $O(n)$。若希望从表中快速确定一个结点的直接前趋，可以在单链表的每个结点里再增加一个指向其直接前趋的指针域 prior。这样形成的链表中有两条方向不同的链，故称之为双(向)链表。

双向链表形式描述为：

```
class DLink{
   private object element;
   private DLink next;
   private DLink prev;
   DLink(object it,DLink n,DLink p)
   {element =it;    next=n; prev=p;}
   DLink(DLink n,DLink p)   { next=n;   prev=p;}
   DLink next(){return next;}
   DLink setNext(DLink nextval){return next=nextval;}
   DLink prev(){return prev;}
   DLink setPrev(DLink prevval){return prev=prevval;}
   Object element(){ return element;}
   object setElement(object it){return element=it;}
   }//class Dlink
```

双向链表的结点结构如图 2-11 所示。

图 2-11 双向链表的结点结构图

其中，element 表示结点的数值；prev 表示指针，指向该结点的直接前趋结点；next 表示指针，指向该结点的直接后继结点。

双向链表的逻辑结构图如图 2-12 所示。

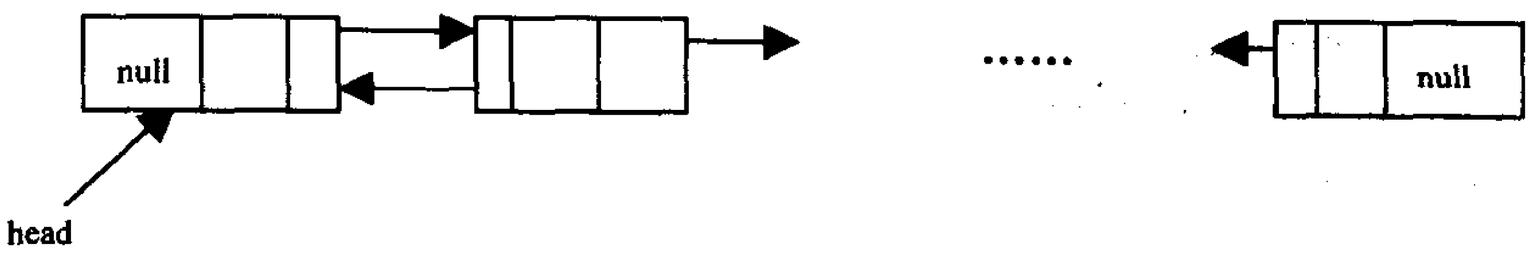


图 2-12 双向链表逻辑图

双向链表的运算类似单向链表的运算，主要包括插入结点、删除结点、查询结点等。

1. 双向链表的插入结点

算法分析：双向链表插入结点的实现比较简单，基本同单向链表，只不过在某个结点前或后插入新的结点时，只要找到该结点就可以了。另外在第一个结点之前插入时同单向链表插入结点，需要单独处理。在插入过程中和单向链表的主要不同点是修改的指针的个数不同。

- 单向链表修改两个指针；
- 双向链表需要修改四个指针。

在链表的 curr 结点后插入一个新结点的逻辑图如图 2-13 所示。

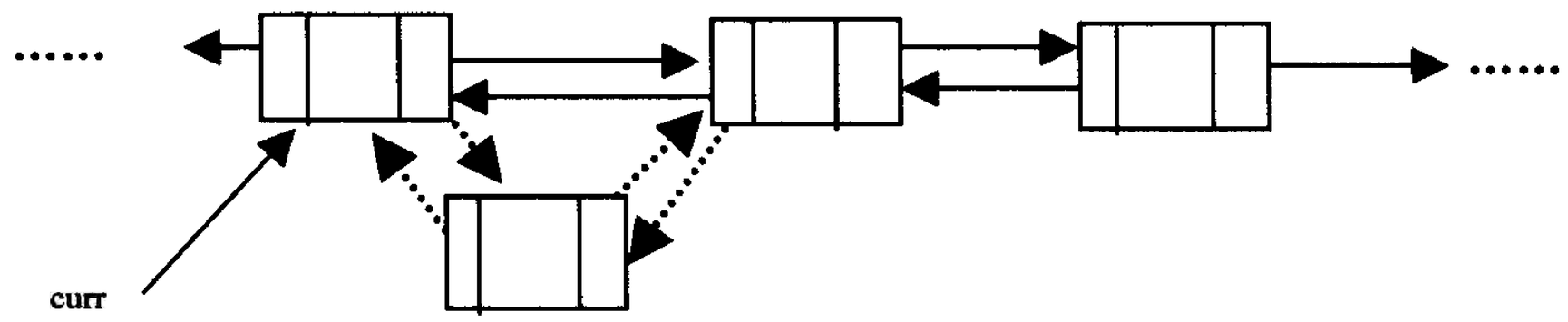


图 2-13　curr 结点后插入结点逻辑图

插入结点应分为以下几个步骤：

(1) 申请新结点，同时给新结点的数据域、两个指针域赋值；

(2) 将当前结点的 next 域指向新结点；

(3) 将当前结点的直接后继的前趋指针指向新结点。

三个步骤可以通过以下的两条语句实现：

```
curr.setNext(new Link(it,curr.next(),curr));//将当前结点的 next 域指向新结点，同时新结点的值为
                                              it，而两个指针域分别是当前结点和当前结点的直接后继结点
if(curr.next().next()!=null)                  //当前结点的直接后继不空
curr.next().next().setPrev(curr.next());//修改后继结点的前趋指针
```

在双向链表 curr 结点的后面插入一个新结点的算法(假设双向链表已经初始化并且已经找到 curr 结点)：

```
//Insert object at current position
 public void insert(object it)
{
 Assert.notNull(curr,"No current element");
 curr.setNext(new DLink(it,curr.next(),curr));
 if (curr.next().next()!=null)
 Curr.next().next().setPrev(curr.next());
 if(tail==curr)
 tail=tail.next();
}
```

如果在双向链表的末尾插入新结点可以直接通过以下函数实现：

```
public void append(object it)
{
tail.setnext(new Dlink(it,null,tail));
tail=tail.next;
}
```

2. 双向链表的删除结点

算法分析：双向链表删除结点的实现基本同单向链表，只不过在某个结点前或后删除新的结点时，只要找到该结点就可以了，不需要保留前趋结点。另外在删除第一个结点时同单向链表删除头结点，需要单独处理。在删除过程中和单向链表的主要不同点是修改的指针的个数不同。

- 单向链表修改一个指针；
- 双向链表需要修改两个指针。

在双向链表的 curr 结点后删除一个结点的逻辑图如图 2-14 所示。

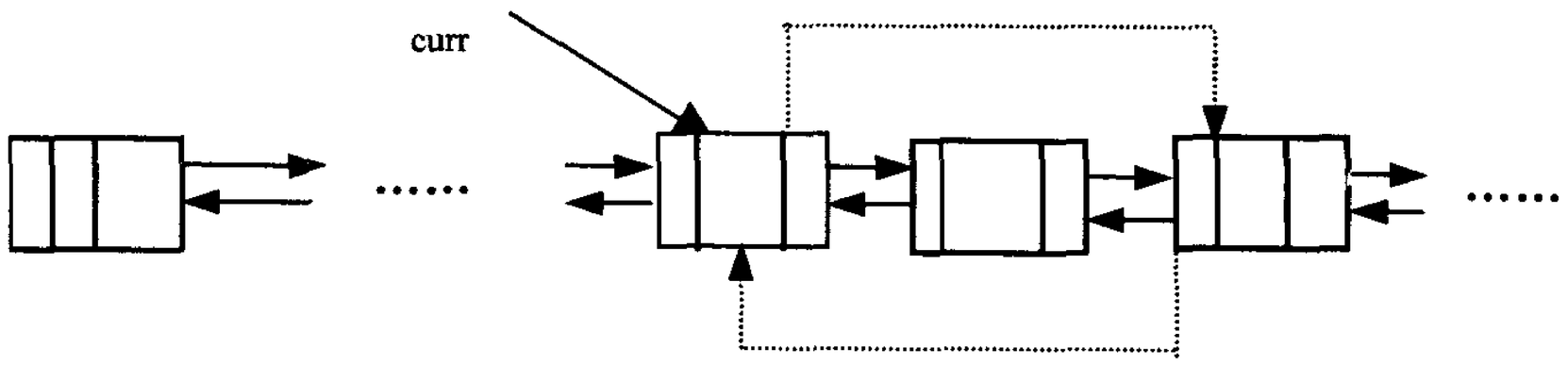


图 2-14 删除结点逻辑图

删除双向链表的结点步骤有：

(1) 修改当前结点的 next 域；

(2) 修改当前结点的后继结点和前趋结点指针。

以上两步可以通过以下的语句实现：

```
if (curr.next().next()!=null)
   curr.next().next().setPrev(curr);
   curr.setnext(curr.next().next());
```

删除结点算法为：

```
//remove object
public void remove ()
{
   Assert.notfalse(isinlist(),"No current element");
   Object it=curr.next().element();
  if (curr.next().next()!=null)
   curr.next().next().setPrev(curr);
```

```
        else tail=curr;
        curr.setnext(curr.next().next());
        return it;
    }
```

将当前结点 curr 前移的函数是：

```
public void prev()
{ if (curr!=null) curr=curr.prev();}
```

3. 双向链表的查询结点

算法分析：双向链表查询结点的实现基本同单向链表，只要按照 next 的方向找到该结点就可以了，不需要保留前趋结点。如果找到，返回结点位置，否则返回 null。

查找结点的步骤是：从头结点开始，直到找到该结点或是查找失败。

查找算法为：

```
public void dlinksearch()
{ curr=head;
 while ((curr!=null)&&(curr.element()!=key))
{curr=curr.next;
 }
if (curr==null)
return null;
else
   return curr;
}
```

和单链表类似，双链表一般也是由头指针 head 惟一确定的，有时为了处理双向链表方便(不需要特殊处理头结点)，在链表上增加一个人为设置的头结点，此结点的值是一个无效值，有了它能使双链表上的某些运算变得方便。

双向链表与单向链表相比的缺点就是使用的空间更多，因为双向链表每个结点需要两个指针的开销。它需要的结构性开销是单向链表的两倍。

如果将双向链表头结点的前趋指针指向链表的最后一个结点，而末尾结点的后继指针指向第一个结点，此时所有的结点链接起来也构成循环链表，称之为双(向)循环链表。

双向循环链表的各种算法与双向链表的算法大同小异，其区别与单链表和单向循环链表的区别一样，就是判断末尾结点的条件不同。双向链表的末尾结点后继指针域为空，而双向循环链表的末尾结点的后继指针域指向第一个结点；而反向查找时，双向链表的头结点前趋指针域为空，而双向循环链表的头结点的前趋指针域指向最后一个结点。

2.4 链表应用举例

例 1：链式存储下的一元多项式加法。

在数学中，符号多项式就是形如 ax^e 的项之和。换句话说，一个一元 n 次多项式按降序排列，可以写成：

$$A_n(x)=a_nx^n + a_{n-1}x^{n-1} + \cdots\cdots + a_1x + a_0$$

当 $a_n \neq 0$ 时，称 $A_n(x)$为 n 阶多项式。其中 a_n 为首项系数。因此一个 n 阶标准多项式由 $n+1$ 个系数惟一确定。在数据结构中，一个 n 阶多项式可以用线性表表示为：

$$A=(a_n, a_{n-1}, \cdots\cdots, a_1, a_0)$$

可以在计算机内部对 A 采用顺序存储结构，使多项式的某些运算变得更简洁。但是，实际情况中的多项式的阶数可能很高，而且不同的多项式阶数可能相差很大，这使顺序存储结构的最大长度难以确定。也就是说，若多项式的阶数很高，并且最高次幂项与最低次幂项之间缺项很多(即系数为零的项很多)时，如：

$$A(x)=x^{1000} + 5$$

若采用顺序存储结构显然十分浪费存储空间。因此一般情况下多采用链式存储存储多项式。

在用线性链表存储一个多项式时，每个系数非零项对应一个具有三个域的结点，结点的结构如图 2-15 所示。

图 2-15 多项式结点结构图

其中，coef 用来表示存放某一项的系数；exp 用来表示存放某一项的指数；link 用来表示存放指向该项的下一项所在结点的指针。

例如，$S(x) = 6x^5 - 4x^3 + 5$ 可以表示成图 2-16 的形式。

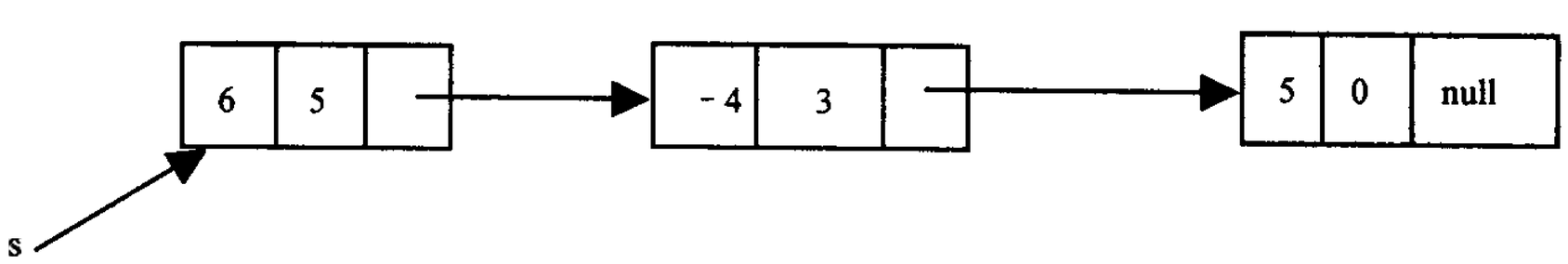


图 2-16 多项式 S 的链式表示图

下面讨论链式存储结构下多项式的相加运算，描述相加运算时，可以作图来说明。

假设 $B_m(x)$为一元 m 阶多项式，则 $B_m(x)$与 $A_n(x)$的相加运算 $C_n(x)=A_n(x)+B_m(x)$ (设 $n>m$) 用线性表表示为：

$$C=(a_n, a_{n-1}, \cdots\cdots, a_{m+1}, a_m+b_m, a_{m-1}+b_{m-1}, \cdots\cdots, a_0+b_0)$$

采用链式存储结构分别表示为三个一元多项式。其中，*A*、*B*、*C* 分别为指向各链表第一个结点的指针。

算法分析：一元多项式加法运算很简单：两个多项式中所有指数相同的项对应系数相加，若和不为零，则构成“和多项式”中的一项，而所有指数不相同的那些项均复制到“和多项式”中。

算法中设置了两个活动指针变量 *pa* 和 *pb*，它们分别沿着 *A* 链表与 *B* 链表依次访问各自链表中的结点。*pa* 的初值为 *A*，*pb* 的初值为 *B*，即分别指向各自链表的第一个结点。

整个算法的核心是：比较 *pa* 和 *pb* 所指结点的 exp 域的值，若相同，则将两结点的系数域值相加，相加的结果不为 0，则把这个结果和相应指数分别存入 *C* 链表中新申请到的空结点的系数域和指数域；若 *pa* 与 *pb* 所指的结点指数不同，先将指数较高的那一项复制到 *C* 链表中，然后将 *pa* 或者 *pb* 移向下一个结点。重复上述步骤，当 pa=null(或 pb=null)时，就将 *B* 链表(或 *A* 链表)剩余部分复制到 *C* 链表中，直到 *pa*、*pb* 均为空时算法结束。

将多项式的链表结构定义为：

```
class link {
   private link next;
   private object data1;
   private object data2;
   link(object it , object it1,link nextval)
   { data1=it ;data2=it1;next=nextval;}
   link(nextval){next=nextval;}
   link next(){return next;}
   link setnext(link nextval){return next=nextval;}
   object data1(){return data1;}
   object setdata1(object it){return data1=it;}
   object data2(){return data2;}
   object setdata2(object it){return data2=it;}
}
```

根据以上的定义，假设 *A*、*B* 两个一元多项式已经存在(可以采用建立单向链表的方法建立两个单链表)。两个多项式的和放在 *C* 链表中，表中的头结点为 head。

多项式相加的算法为：

```
private void setup()
{tail=head=curr=new link(null);
  }
 public void expadd()
 { link pa=A;
    link pb=B;
    int x=link pa.next().data1;
    int y=link pb.next().data1;
```

```
while ((pa!=null)&&(pb!=null))
 { if (x<y)
  {link curr.setnext(new Link(pb.next().data1,pb.next().data2,head));
   head=curr;
   pb=pb.next();}
    else
       if (x>y)
       {link curr.setnext(new Link(pa.next().data1,pa.next().data2,head));
        head=curr;
        pa=pa.next();}
      else
        {int z=pa.setdata2()+pb.setdata2();
         if (z!=0)
         {
        link curr.setnext(new Link(pa.next().data1,z,head));
        head=curr;
     }
        pa=pa.next();
        pb=pb.next();
     }
while (pa!=null)
        {
       link curr.setnext(new Link(pa.next().data1,pa.next().data2,head));
       head=curr;
            pa=pa.next();
            }
          while (pb!=null)
  {
 link curr.setnext(new Link(pb.next().data1,pb.next().data2,head));
 head=curr;
       pb=pb.next();
       }
return head;
 }
```

例 2：Josephus 问题。

Josephus 问题是建立在历史学家 Joseph ben Matthias(称为 Josephus)的一个报告的基础之上，该报告讲述了他和 40 个士兵在公元 67 年被罗马军队包围期间签定的一个自杀协定，Josephus 建议每个人杀死他的邻居，他很聪明地使自己成为这些人中的最后一个，因此获得生还。假设对该问题使用 8 个士兵时的情况进行模拟。

首先应该定义存储结构，假设使用双向循环链表解决，定义一个双向循环链表，且为 Ring 类，是线性表的一个子类，其中 Ring 类的定义为：

```
public class Ring extends java.util.AbstractSequentialList
{  private  Node  header;
   private  int  Size  =  0;
   public  Ring()
   {
   }
   public  Ring(List  list)
   {  super();
addall(list);
}
public  ListIterator  listIterator(int  index)
{  return  new  RingIterator(index);
)
public  int  size()
{  return  size;
}
private  static  class  Node
{  object  object;
Node  previous, next;
Node(object  object,  Node  previous,  Node  next)
{  this.object = object;
this.previous = previous;
this.next = next;
}
Node(object  object)
{  this.object = object;
this.previous = this.next = this;
}
}
private  class  RingIteratOr  implements  ListIterator
{  private  Node  next,  lastReturned;
private  int  nextIndex;
RingIterator(int  index)
{  if  (index<0  || index>size)
   throw  new  IndexoutOfBoundsException("Index  :  "  +  index);
next=(size==0?null:header);
for  (nextIndex=0;  nextIndex<index;  nextIndex++)
nex = next.next;
}
public  boolean  hasNext()
{  return  size  >  0;
}
public  boolean  hasPrevious()
```

```
{  return  size  >  0;
}
public  object  next()
{  if  (size==0)  throw  new  NoSuchElementException();
lastReturned  =  next;
next  =  next.next;
nextIndex  =  (nextIndex==size-1?0:nextIndex+1);
return  LastRetUrned.object;
}
public  object  previous()
{  if  (size==0)  throw  new  NoSuchElementException();
next  =  lastReturned  =  next.previous;
nectIndex  =  (nextIndex==0 ?size-1:nextIndex-1);
return  lastReturned.object;
}
public  int  nextIndex()
{  return  nextIndex;
}
public  int  previousIndex()
{  return  (nextIndex==0 ? size-1:nextIndex-1);
}
public  void  add(object  object)
{  if  (size==0)
{  next = header = new  Node(object);
nextIndex = 0;
}
else
{  Node  newNode = new  Node(object, next.previous, next);
newNode.previous.next = next.previous = newNode;
}
lastReturned = null;
++size;
nextIndex = (nextIndex==size-1 ? 0:nextIndex+1);
}
public  void  remove()
{  if  (lastReturned==null)  throw  new  IllegalStateException();
if  (next==lastReturned)  next = lastReturned.next;
else  nextIndex = (nextIndex==0 ? size-1:nextIndex-1);
lastReturned.previous.next = lastReturned.next;
lastReturned.next.previous = lastReturned.previous;
lastReturned = null;
--size;
}
```

```
    public  void  set(object  object)
    {  if  (lastReturned==null)  throw  new  IllegalStateException();
    lastReturned.object = object;
    }
    }
}
```

在上述程序中，Ring 类是 AbstractSequentialList 类的一个子类，它需要实现 listIterator(int)方法和 size()方法。

算法分析：Josephus 问题在已经定义了 Ring 类后，可以简单考虑为一个循环链表，从某一点开始，逐步删除相邻的结点，直到最后剩余一个结点为止。

```
public class joseph
{ public static void main(string[]   args)
   {Ring   ring=new Ring();
    listiterator it=ring.listiterator();
    int N=get("enter    number   of   soldiers");
    for (int k=0;k<N;k++)
       it.add(new Character((char)( 'A'+k));
        system.out.print(N + "Soldiers: ");
system.out.println(ring);
        while (ring.size()>1)
        {object   killer = it.next();
         system.out.println(killer + "killed" + it.next());
 it.remove();
}
system.out.println("the   lone   survivor   is "+ it.next());
}
Public static int get(String     prompt)
{int   n=0;
 try
{Inputstreamreader     reader=new Inputstreamreader(system.in);
 Bufferedreader   in=new   bufferedreader(reader);
 System.out.print(prompt + ":");

 String   input=in.readline();
 n = integer.parseint(input);
}
catch(exception     e)
 {system.out.println(e);}
return   n;
}
}
```

例 3：使用迭代器编写一个将链接线性表逆序打印的算法。

实现线性表逆序的方法有很多种，如果使用迭代器逆序打印实现起来比较容易。首先迭代器是一个对象，它能够从集合中的一个元素移至下一个元素，它是数组或向量下标的另一个选择。例如，一个双向迭代器的定义为：

```
public interface Listiterator extends   Iterator
{ public void add(Object   object);
  public boolean   hasnext();
  public boolean   hasprevious();
  public Object next();
  public int   nextindex();
  public Object previous();
  public int   previousindex();
  public void remove();
  public void   set(Object   object);
  }
```

迭代器的使用为：

```
public class diedaiqi
{ public static void main(String[]   args)
{String[] planets=new String[]{"Venus","Earth","Mars","Pluto"};
List list=Arrays.asList(planets);
System.out.println("list ="+ list);
ListIterator   it= list.listIterator();
System.out.println("it.next() =" + it.next());
System.out.println("it.next() =" + it.next());
System.out.println("it.next() =" + it.next());
System.out.println("it.next() =" + it.next());
System.out.println("it.previou() =" + it.previous());
System.out.println("it.previou() =" + it.previous());
}
}
```

使用迭代器逆序打印实现为：

```
public static void printforward(LinkedList   list)
{ ListIterator   itr=list.listIterator(list.size());
  while (itr.hasprevious())
    system.out.println(itr.previous());
}
```

2.5　顺序表和链表的比较

线性表的存储有两种：顺序存储表和链式存储表。具体存储方式可根据具体问题的要求和性质来决定。

根据线性表定长与不定长确定：顺序存储结构一般要求数据存放的物理和逻辑地址连续；而链式存储结构数据存放地址可连续可不连续，在线性表长度没有确定的情况下，一般采用链式存储结构比较好，反之应以顺序存储为主。

在实际应用中，考虑何种存储结构，可以根据具体的情况具体分析。一般选择存储结构时可以主要从以下两个方面考虑。

(1) 基于空间的考虑

顺序表的存储空间是静态分配的，在程序执行之前一般必须明确规定它的存储规模。若线性表的长度 n 变化较大，则存储规模难于预先确定(定义太大可能浪费空间，定义太小又可能不够用)。因此，当线性表的长度变化较大，难以估计其存储规模时，以采用动态链表作为存储结构为好；反之如果存储规模比较小，并且线性表的长度一般固定时，可使用顺序存储。

存储密度=(结点数据本身所占的存储量)/(结点结构所占的存储总量)

一般地，存储密度越大，存储空间的利用率就越高。顺序表的存储密度为 1，而链表的存储密度小于 1。由此可知，当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构。

(2) 基于时间的考虑

顺序表是由向量实现的，它是一种随机存取结构，对表中任一结点都可在 $O(1)$时间内直接地存取，而链表中的结点需顺序存取，应从头指针起顺着链指针扫描结点才能获得。因此，若对线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表的存储结构比较好。反之，如果在线性表中需要做较多的插入和删除，采用顺序存取，就可能造成大量的数据移动，在时间上的开销较大，而采用链式存储时，只需要修改相应的指针就可以了。所以如果比较偏重线性表的查找，通常很少对线性表进行插入删除操作时，因为顺序存储结构为随机存取(存取速度快)，而链式存储结构为顺序存取(存取速度相对较慢)，此时应采用顺序存储结构较好。

思考和练习

1. 基础知识题

(1) 在什么情况下，顺序表比链表好？

(2) 简述线性表、单链表、线性表的存储方式和双链表、循环链表的定义。

(3) 描述以下 3 个概念的区别：头指针、头结点(人为设置)和首元结点(第一个元素结点)。

(4) 在链表中人为设置头结点的作用是什么？

(5) 为什么在单向循环链表中设置尾指针比设置头指针更好？

(6) 在顺序表中插入或删除一个数据元素，需要平均移动_____个数据元素，移动数据元素的个数与_____有关。

(7) 顺序表中逻辑上相邻的元素的物理位置为_____相邻，单链表中逻辑上相邻的元素的物理位置为_____相邻。

(8) 单链表是由多个结点所串连而成的，每一个节点包含了_____和指向链表的下一个结点的指针。

(9) 使用数组如何实现单链表的指针表示？

(10) 下列哪一个程序片段是在链表中间插入一个节点？(假设新结点为 new，欲插入在 pointer 结点之后)

(a) next[new]=pointer;
    pointer=new;
(b) next[new]=next[pointer];
    next[pointer]=new;
(c) next[pointer]=next[new];
    next[new]=pointer;
(d) 以上皆非。

(11) 下列哪一个程序片段是删除链表中间结点？(假设欲删除结点为 pointer 结点，back 为前一个结点，-2 表示未用空间。)

(a) next[pointer]=-2;
(b) next[back]=-2;
(c) back=next[pointer];
    next[pointer];
(d) next[back]=next[pointer];
    next[pointer]=-2;

(12) 如有一个链表如图 2-17 所示。

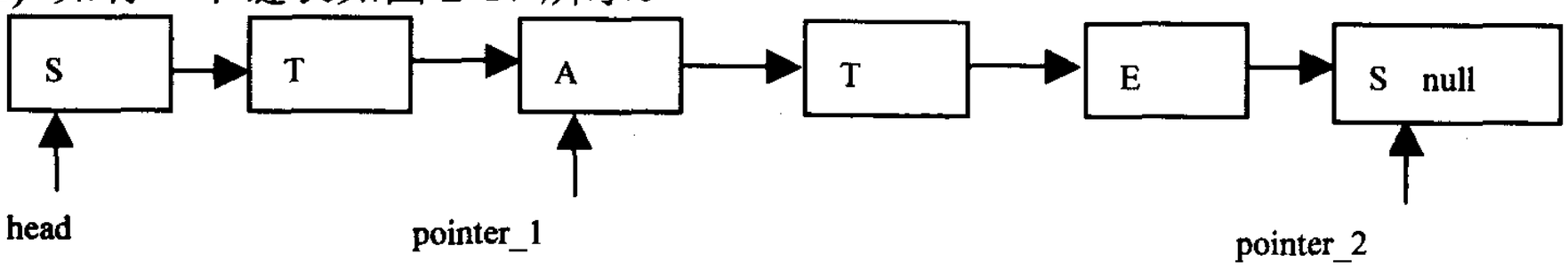


图 2-17

下列语句哪一个是正确的？

(a) data[next[HEAD]]= "A";
(b) data[HEAD]= "T";
(c) data[pointer_1]= "A";
(d) next[pointer_2]=NULL

2. 算法设计题

(1) 编写一些 Java 语句，用线性表建立一个能存放 20 个数据元素而实际只存放(2，23，5，15，47)序列的线性表。

(2) 设计将一个双向循环链表逆置的算法。

(3) 写出在一个头结点的单链表中的值为 x 的结点之后插入 m 个结点的算法。

(4) 编写建立一个单向循环链表算法。

(5) 编写建立一个双向循环链表算法。

(6) 编写算法对无序的线性表实现有序排列。

(7) 试编写在不带头结点的单链表上实现线性表基本运算 LENGTH(L)的算法。

(8) 假设有两个按数据元素值递增有序排列的线性表 A 和 B，均以单链表作存储结构。编写算法将 A 表和 B 表归并成一个按元素值递减有序(即非递增有序，允许值相同)排列的线性表。

(9) 已知单链表 L 中的结点是按值非递减有序排列的，试写一算法将值为 x 的结点插入表 L 中，使得 L 仍然有序。

(10) 试分别以顺序表和单链表作存储结构，各写一个实现线性表的就地(即使用尽可能少的附加空间)逆置的算法，在原表的存储空间内将线性表(a_1，a_2，……，a_n)逆置为(a_n，a_{n-1}，……，a_2，a_1)。

(11) 假设在长度大于 1 的循环链表中，无头结点也无头指针。S 为指向链表中当前结点的指针，试编写算法删除结点的前趋结点。

(12) 已知一单链表中的数据元素含有 3 类字符(即字母字符、数字字符和其他字符)。试编写算法，构造 3 个循环链表，使每个循环链表中只含同一类的字符，且利用原表中的结点空间作为这 3 个表的结点空间(头结点另辟空间)。

3. 应用题

(1) 试利用单链表编写一个学生成绩系统，该系统具有查询成绩、修改成绩、删除成绩、新增成绩、全班平均成绩等功能。

数据如表 2-1 所示。

表 2-1　学生成绩表

学生座号	学生姓名	语文成绩	英语成绩	数学成绩
6	张力	85	90	98
15	梁宽	76	70	80
17	盛行	95	98	96
20	李蓝	65	60	80
23	刘文	79	65	86
32	陈玫	93	86	74

(2) 用无序线性表实现一个城市数据库。每条数据库记录包括城市名(任意长的字符串)和城市的坐标(用整数 x 和 y 表示)。你的数据库应该允许插入记录、按照名字或者坐标删除或检索记录，还应该支持打印在指定点给定距离内的所有记录。先使用顺序表实现，然后用链表实现。记录用这两种方法实现的每次操作的运行时间。这两种实现方法的相对优、缺点是什么？如果按照城市名的字母顺序来存储记录，使得线性表成为有序的，这样能加速一些操作吗？这种按照城市名排序的线性表会减慢一些操作吗？

(3) (Josephus 环)任意正整数 n、k，按下述方法可得排列 1，2，……，n 的一个置换：将数字 1，2，……，n 环形排列，按顺时针方向从 1 开始计数，计满 k 时输出该位置上的数字(并从环中删去该数字)，然后从下一个数字开始继续计数，直到环中所有数字均被输出为止。例如，当 n=10，k=3 时，输出的置换是 3，6，9，2，7，1，8，5，10，4 等。试写一算法，对输入的任意正整数 n、k 输出相应的置换。

(4) 利用单链表循环结构编写多项式相乘的算法。$C=A*B$，不要破坏 A、B 表，将 C 表示成一个新表。

(5) 编写一个通用的类，用于从一个 16 位二进制数中取出偶数位后生成 8 位二进制数并将其转换为十进制数。

第3章　栈 和 队 列

堆栈和队列是两种特殊的线性表，学习本章的内容主要充分理解堆栈和队列的运算规则。堆栈的主要特点是只能在栈顶操作，也就是遵循先进后出的运算规则。队列的主要特点是只能在一端插入、另一端删除，也就是遵循先进先出的运算规则。本章主要通过学习堆栈和队列，理解队列和堆栈在实际中的应用。

本章的学习目标：

- 堆栈定义和堆栈的运算规则；
- 堆栈的基本运算；
- 队列的定义和运算规则；
- 队列的基本运算，特别是循环队列的入出队的运算。

3.1　栈

3.1.1　栈定义及基本概念

栈(Stack)又称堆栈，是限制在表的一端进行插入和删除运算的线性表。通常称能够进行插入、删除运算的这一端为栈顶(Top)，另一端称为栈底(Bottom)。当表中没有元素时称为空栈。

习惯上将每次删除(也称为退栈)操作又称为弹出(Pop)操作。删除的元素总是当前栈中“最新”的元素(栈顶元素)；将每次插入(称为进栈)操作称为压入(Push)操作，压入的元素总是当前栈中“最新”的元素。

在空栈中最先插入的元素总被放在栈的底部，只有所有元素被弹出之后它才能被删除。

当栈满时进栈运算称为“上溢”；当栈空时退栈运算称为“下溢”。

堆栈的存储结构有顺序存储结构和链式存储结构两种，在顺序存储结构下，可以考虑堆栈的上溢，而在链式存储结构下，不必考虑堆栈的上溢现象，只需要考虑堆栈的下溢现象。

堆栈上溢是一种出错状态，应该设法避免它；而下溢则可能是正常现象，通常下溢用来作为程序控制转移的条件。

堆栈的运算是按后进先出的原则进行的(又称为先进后出或后进先出)，简称为LIFO(last in first out)表(所有的运算只能在栈顶运行如图 3-1 所示)。就线性表而言，实现栈

的方法有很多，我们着重介绍顺序栈(arrary-based stack)和链式栈(linked stack)，它们类似于顺序表和链式表。

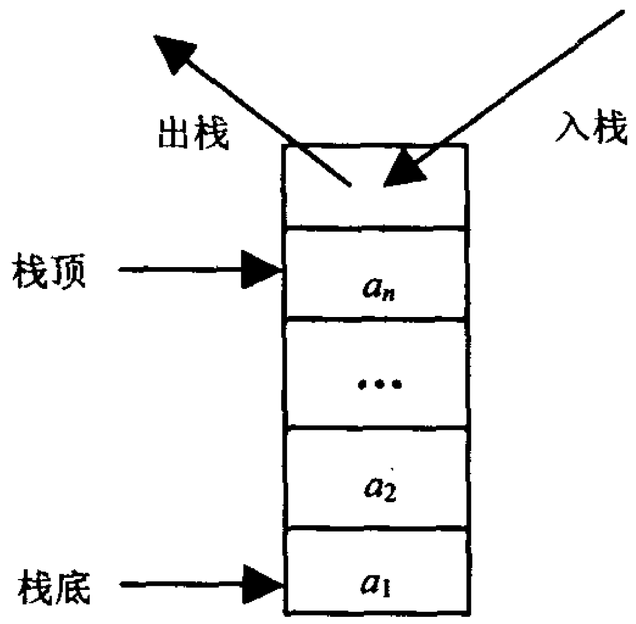


图 3-1　堆栈示意图

栈的基本运算一般有以下几种：

- InitStack(S)　构造一个空栈 *S*。
- StackEmpty(S)　判栈空，若 *S* 为空栈返回 TRUE，否则返回 FALSE。
- StackFull(S)　判栈满，若 *S* 为满栈，则返回 TRUE，否则返回 FALSE。该运算只适用于栈的顺序存储结构。
- Push(S, x)　进栈。若栈 *S* 不满，则将元素 *x* 压入 *S* 的栈顶。
- Pop(S)　退栈。若栈 *S* 非空，则将 *S* 的栈顶元素弹出，并返回该元素。
- StackTop(S)　取堆栈的栈顶元素，不修改栈顶指针。

比较重要的运算就是入栈和出栈两种。当然每一种运算对应不同的存储结构实现的方法可能有所不同。

抽象数据类型定义堆栈如下：

```
ADT Stack{
    数据对象:D={ai|ai∈elementset,i=1,2,3,……,n, n≥0 }
    数据关系:R={<ai-1,ai> | ai-1,ai∈D,约定 an 端为栈顶，a1 端为栈底}
    基本操作:
    initstack(s);              //初始化，结果：构造一个空栈
    clearstack(s);             //清空堆栈;
    stackempty(s);             //判断堆栈是否为空;
    stackfull(s);              //判断栈满;
    gettop(s);                 //取栈顶元素;
    push(s,x);                 //压入堆栈一个元素;
    pop(s);                    //从栈顶弹出一个元素;
    stacklength(s);            // 计算堆栈中元素的个数;
}ADT stack
```

根据堆栈的不同存储结构，可以实现对堆栈的不同操作。堆栈是一种特殊的线性表，所以堆栈的存储结构一般也有两种：顺序存储结构和链式存储结构。

压入元素称为入栈，实际上是在线性表的头部插入一个元素；弹出一个元素称为出栈，

实际上是在线性表的头部删除一个元素。

3.1.2　顺序栈

顺序栈的实现从本质上讲，就是顺序线性表实现的简化。惟一重要的是需要确定应该用数组的哪一端表示栈顶。一种选择是把数组的第 0 个位置作为栈顶。根据线性表的函数，所有的插入(insert)和删除(remove)操作都在第 0 个位置的元素上进行。由于这时每次 push(insert)或者 pop(remove)操作都需要把当前栈中的所有元素在数组中移动一个位置，因此效率不高。如果栈中有 n 个元素，则时间代价为 $O(n)$。另一种选择是当栈中有 n 个元素时把位置 $n-1$ 作为栈顶。也就是说，当向栈中压入元素时，把它们添加到线性表的表尾，成员函数 pop 也是删除表尾元素。在这种情况下，每次 push 或者 pop 操作的时间代价仅为 $O(1)$。

堆栈的运算可以用图的形式简单地描述。

主要考虑堆栈的入栈和出栈算法。其原因是在堆栈的基本运算中有 6 种：判断堆栈空、堆栈初始化、判断堆栈满(仅限于顺序存储的情况下)、入栈元素、出栈元素、取栈顶元素等。而入栈时需要考虑的操作步骤是堆栈初始化，然后判断堆栈是否为满，如果不满，则可以插入元素(入栈)；出栈时，需要考虑的步骤是判断堆栈是否为空，如果不空，删除元素(出栈)，出栈之前，保存栈顶元素。也就是说，堆栈的入出栈运算包含了其他的 6 种基本运算，取栈顶元素的运算，只是不用修改栈顶的指针而已。

入栈操作如图 3-2 所示。

假设将 A、B、C 依次压入堆栈形式如图 3-3 所示，数组的容量为 3。

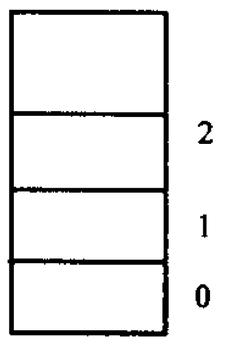


图 3-2　堆栈的初始状态

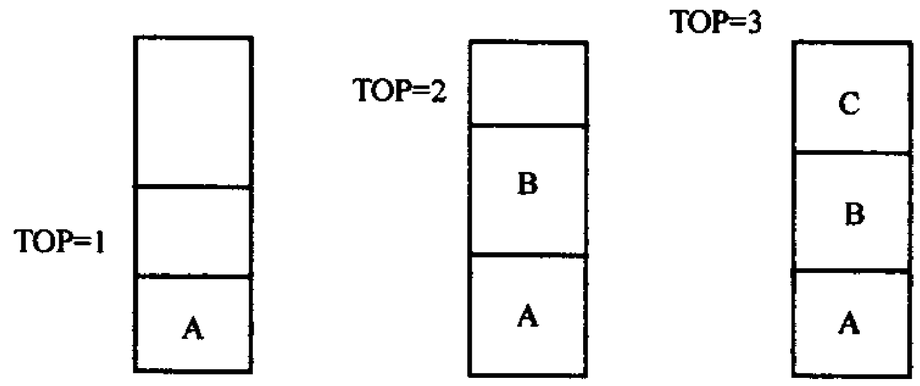


图 3-3　堆栈压入元素的逻辑图

弹出顺序依次如图 3-4 所示。

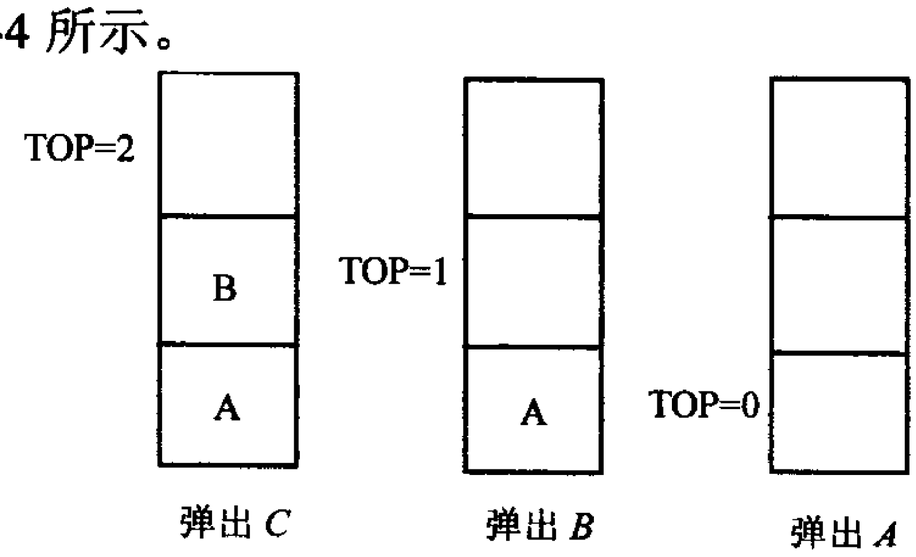


图 3-4　堆栈弹出元素的逻辑图

从以上的压入和弹出结构可以看出，堆栈的运算规则是先进后出。在堆栈的运算中，无论是何种存储结构，只要定义了堆栈，对它的各种操作就必须严格按照堆栈的运算规则执行。

根据先进后出的原则，顺序栈的基本运算为：

```
Class astack {//array based stack class
    private static final int defaultsize=10;
    private int size;                  //maxinum size of stack
    private int top;                   //Index for top object
    private object[] listarray;        //Array holding stack objects
    //定义堆栈的空间大小
    AStack() {setup(defaultSize);}
    AStack(int sz) {setup(sz);}
    //堆栈初始化
    public void setup(int sz)
    { size=sz; top=0;listarray=new object[sz];}
    //堆栈清空
    public void clear()
    {top=0;}
    //栈顶压入元素
    public void push(object it)
    {
    Assert.notFalse(top<size,'Stack overflow');
    listarray[top++]=it;
    }
    //弹出栈顶元素
    public object pop()
    {
    Assert.notFalse(!isEmpty()，'empty stack');
    return listarray[--top];
    }
     //取栈顶元素值
    public object topValue()
    {
    Assert.notFalse(!isempty(),'empty stack');
    return listarray[top-1];
    }
     //测试堆栈是否为空
    public boolean isempty()
     { return top==0;}
     }//class AStack
```

在此堆栈类中，top 值保留的是一个即将插入元素的位置。如果 top 保留最后插入的元素的位置，初始情况下，堆栈的值应是 - 1。此时 PUSH 运算首先应将 top 加 1 后，才能插入元素，而不是像此例中先放元素再加 1。同样 POP 运算应先保留元素，然后 top - 1，而不是像此例中先减 1。

堆栈顺序存储时，为避免上溢，需要首先分配较大空间，但这容易造成大量的空间浪费。所以当使用两个栈时，可以将两个栈的栈底设在向量空间的两端，让两个栈各自向中间靠拢，使空间得以共享。值得注意的是：此种情况下，虽然减少了空间的浪费，但同时也增加了堆栈溢出的可能性。因为每个堆栈的可利用空间相应地减少了。

其逻辑图如图 3-5 所示。

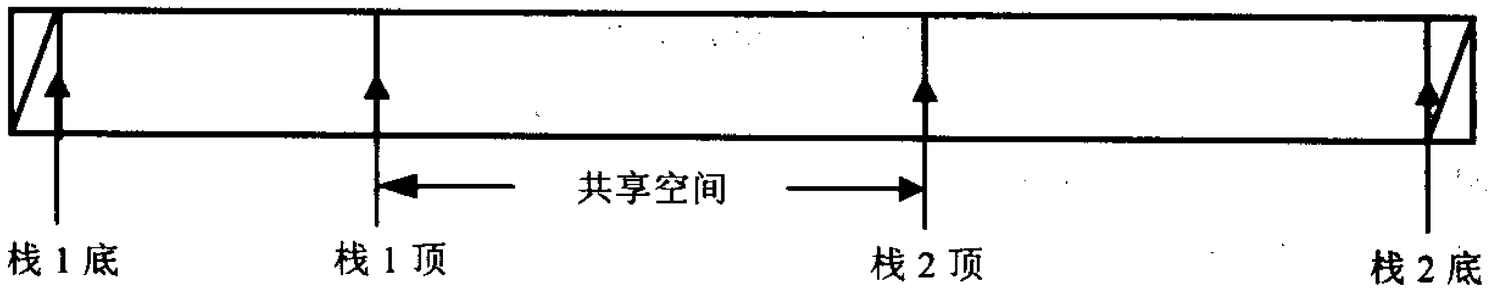


图 3-5　两个堆栈共享空间逻辑图

具体实现的方法是：利用一个数组来存储两个堆栈，每个栈从各自的端点向中间延伸，这样浪费的空间就会减少。也就是说，从一个堆栈取出元素时，栈顶指针是增加的，而另一个堆栈的栈顶指针是减少的；反之压入一个元素时，一个堆栈的栈顶指针是增加的，而另一个堆栈的栈顶指针是减少的。

对于多个堆栈的共享，也可以利用以上的方法，但是当有多个堆栈时，可能会出现有两个堆栈共享的空间已经满了，但是其他堆栈可能还有存储空间。此时需要为有空间的堆栈为满的堆栈提供空间，这要造成大量的数据移动，虽然节省了空间，但也增加了时间的开销，这就需要在实际应用中，衡量得与失，按照实际需要来解决问题。

3.1.3　链式栈

堆栈的链式存储称为链栈(单向链表存储堆栈)。链式栈的基本运算同顺序栈，定义也同线性表的链表定义，它是对链表实现的简单化(因为它只是对链表的头部操作)。它的元素只能在表头进行插入和删除。在链式存储结构中，不需要给出表头结点，因为其中惟一的已知条件是栈顶指针 top，它是指向链式栈的第一个结点(相当于头结点)。

堆栈的各种运算比链式存储的普通线性表运算实现时要方便得多，主要原因是堆栈的各种运算只能在堆栈的一端操作。堆栈是特殊的线性表，我们只要考虑对线性表的一端操作的情况，并且只能在一端插入、删除(栈顶)；而线性表除此之外(不限定一端插入、删除)，还需要考虑另外一端结点以及中间结点的插入、删除、查询等操作，可以把堆栈的入出堆栈运算当作线性表的一端进行插入、删除的特例。

注意：

堆栈的运算一定遵循“先进后出”的运算规则。

对应链式栈的基本运算是：

```
class linkstack{
    private   link top;        //栈顶指针
    //堆栈设置
```

```
 public   linkstack(int sz)
 { setup();
 }
//堆栈初始化
private void setup()
{top=null;}
 //堆栈清空
public void clear()
 {top=null;}
//在堆栈中压入一个元素
public void push(object it)
 {
  top=new link(it,top);} //申请一个新结点，数据域的值是 it，同时，将新结点的指针域指向原
                          来的 top 结点，新结点的指针为 top
 //在堆栈中弹出一个元素
public void pop()
{ assert.notfalse(!isempty(),"empty stack");//堆栈是否为空
   object it=top.element();                    //保留弹出的元素
   top=top.next();                             //修改栈顶指针
   return it;
 }
//取栈顶元素
public object topvalue()
{ assert.notfalse(!isempty(),"not top value");
   return top.element();
 }
//测试堆栈是否为空
 public boolean isempty()
{return top==null;}
 }    //class linkstack
```

其中 push 首先修改新产生的链表结点的 next 域并指向栈顶，然后设置 top 指向新的链表结点；而 pop 中则是用 it 保存栈顶值，top 指向当前栈顶链接到的下一个结点，it 的值作为 pop 的返回值。

值得注意的是：因为堆栈只能在头部(栈顶)操作，所以链式存储堆栈时不能附加表头结点。

3.1.4 顺序栈和链式栈的比较

实现链式栈和顺序栈的操作都是需要常数时间，即时间复杂度为 $O(1)$，主要从空间和时间两个方面考虑。

初始时，顺序堆栈必须说明一个固定的长度，当堆栈不够满时，造成一些空间的浪费，而链式堆栈的长度可变则使长度不需要预先设定，相对比较节省空间，但是在每个结点中设置了一个指针域，从而产生了结构开销。

当需要多个堆栈共享时，顺序存储中可以充分利用顺序栈的单向延伸性。可以使用一个数组存储两个堆栈，使每个堆栈从各自的端点向中间延伸，这样浪费的空间就会减少。但只有当两个堆栈的空间有相反的需求时，这种方法才有效。也就是说，最好一个堆栈增长，一个堆栈缩短。反之，如果两个堆栈同时增长，则可能比较容易造成堆栈的溢出。如果多个顺序堆栈共享空间，且一个共享的堆栈满了，此时可能其他的堆栈没有满，则需要按照堆栈的运算规则(LIFO)，将满栈的元素向右或左平移，这就可能造成大量的数据元素移动，使得时间的开销增大。相对而言，使用两个堆栈共享一个空间是比较适宜的存储方法，但同时也增加了堆栈溢出的可能性。链式堆栈由于存储的不连续性，什么时候需要空间，什么时候就可以申请，一般不存在堆栈满的问题，但是链式存储的堆栈在存储时，需要增加指针开销，从总体而言，比较浪费空间，好处是一般不需要栈的共享。

3.1.5　栈的应用举例

实际生活中先进后出的实例比较多，例如几辆火车进入同一个铁轨，出来时必须遵循先进后出；计算机使用中断时，保护现场使用堆栈，中断返回时从堆栈中弹出(恢复现场)，也是遵循先进后出原则；数学中求阶乘时，可以使用堆栈保存以前的数据，直到 0 的阶乘为止等。堆栈的应用例子比较多，但比较典型的是数制的转换、表达式的计算、转换问题和递归问题。

1. 数制的转换

在十进制数 N 和 d 进制数的转换是计算机实现计算的基本问题，解决的方法有很多，但其中的基本原理是：$N \bmod d$ 的值是余数，余数作为转化后的值，用 $N \operatorname{div} d$ 的商再作为 N 的值，再求余数，依此类推，直到商数为零。最后将所得的余数反向输出，就是我们需要的结果。由此看出，先得到的结果最后输出，而最后得到的结果第一个输出，恰好满足堆栈的运算规则：先进后出。所以，可以使用堆栈实现数制的转换。

```
Import    java.io.*;
Public class    transportnum
{    public static void main(string args[])
{ inputstreamreader    inputreader;
   bufferreader    bufreader;
   inputreader = new inputstreamreader(system.in);
   bufreader = new bufferreader(inputreader);
   tempstr= bufreader.readline();
   int n = integer.parseint(tempstr);
    int n1=0;
```

```
while (n!=0)                    //只要商没有等于 0
  {n1=n%d;                      //所求的余数
   n=n/d;                       //n 中存放除以 d 的余数
  top=new link(n1,top);}        //申请一个新结点，数据域的值是 n1，同时，将新结点的指针域指向
                                 原来的 top 结点，新结点的指针为 top
}
while (!isempty())                      //堆栈不空
  system.out.print(top.element);    //输出元素
}
}
```

2. 表达式的转换

表达式一般有中缀表达式、后缀表达式和前缀表达式3种表示形式，通常我们使用中缀表示，但是中缀表达式在计算机中存储计算时，比较麻烦，所以计算机内存储表达式时，一般采用后缀或前缀表示较多。

一个表达式通常由操作数、运算符及分隔符所构成。一般我们习惯使用中缀描述法，也就是将运算符放在操作数中间，例如：

a+*b***c*

由于运算符有优先级，所以在计算机内部使用中缀描述是非常不方便的，特别是带有括号时更麻烦。为方便处理起见，一般需要将中缀的表达式利用堆栈转换成为计算机比较容易识别(没有括号)的前缀或后缀表达式，这样在扫描表达式时，只要按照运算符直接计算即可。例如：

+*a***bc*	前缀表达式
*abc**+	后缀表达式

其转换的过程按照优先级转换，运算符的优先级如表3-1所示。

表3-1　运算符优先级顺序表

优　先　级	运　算　符
高 ↓ 低	括号：“(”，“)”
	负号：“－”
	乘号：“*”，除号：“/”，余数：“%”
	加号：“+”，减号“－”
	关系：“<”，“>”，“>=”，“<=”，“=”

下面以中缀表达式 *a*/(*b* - *c*)为例说明中缀表达式转换为前缀表达式的具体步骤(先处理优先级高的式子)。

步骤 1：先处理优先级最高的，括号内将(*b* - *c*)转换为(- *bc*)。

步骤 2：将除号进行处理为/a，整个表达式为/*a*(- *bc*)。

步骤 3：消除括号为/*a* - *bc*，就是将中缀变为的前缀表达式。

计算机处理中缀表达式时，假设按照从左向右的扫描顺序，依次存放表达式的各个符号，在扫描过程中，可以一边输入，一边计算。但是因为中缀表达式中操作符的优先级的限制，对新输入的运算符的优先级需要进行比较，然后才能够计算，所以必须使用堆栈保存以前的数据和运算符。

利用堆栈处理中缀表达式的步骤是：

(1) 设置两个堆栈，一个操作数堆栈和一个运算符堆栈。

(2) 初始时为空，读取表达式时，只要读到操作数，将操作数压入操作数栈。

(3) 当读到运算符时将新运算符和栈顶运算符的优先级比较，如果新运算符的优先级高于栈顶运算符的优先级，将新运算符压入运算符堆栈；否则取出栈顶的运算符，同时取出操作数堆栈中的两个操作数进行计算，计算结果压入操作数堆栈。

(4) 重复(2)、(3)步骤，直到整个表达式计算结束为止，此时操作数堆栈中的结果就是表达式的计算结果。

例如：*A*+*B*/*C* - *D*。

步骤 1：设置两个堆栈，如图 3-6 所示。

步骤 2：压入 *A* 到操作数堆栈，如图 3-7 所示。

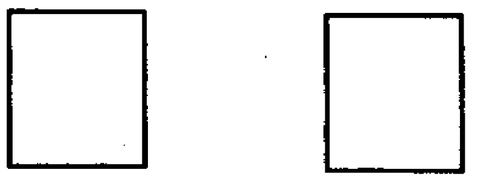


图 3-6　堆栈的初态

图 3-7　扫描“A”时堆栈的状态

步骤 3：将“+”压入运算符堆栈，将 *B* 压入操作数堆栈，如图 3-8 所示。

步骤 4：当读到“/”时，运算符比较优先级。“/”优先级高于栈顶“+”的优先级，所以将“/”压入运算符堆栈，如图 3-9 所示。

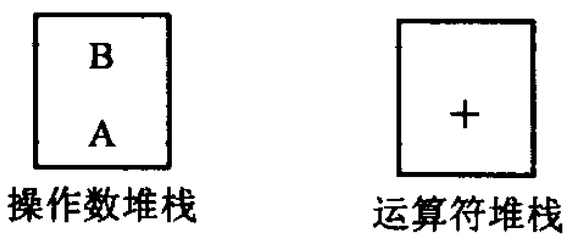


图 3-8　扫描“+B”时堆栈的状态

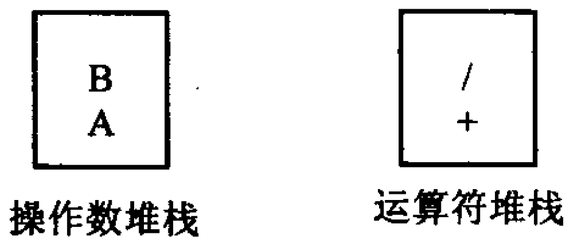


图 3-9　扫描“/”时堆栈状态

步骤 5：读到 *C*，将 *C* 压入操作数堆栈，读到“ - ”时，将“ - ”优先级和栈顶“/”的优先级比较，小于“/”优先级，先弹出“/”，再从操作数堆栈中弹出两个操作数，计算后结果压回操作数堆栈，然后将“ - ”压入运算符堆栈，读到 *D*，将 *D* 压入操作数堆栈，如图 3-10 所示。

步骤 6：取出“ - ”和两个操作数，计算“ - ”，结果压入操作数堆栈，如图 3-11 所示。

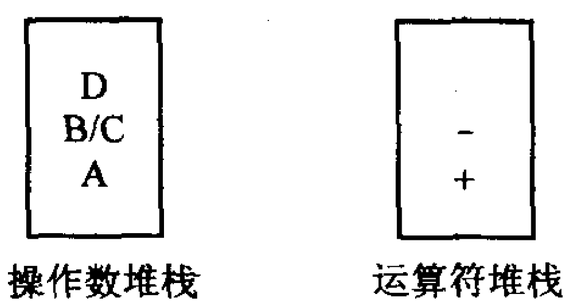


图 3-10 扫描“$C-D$”时堆栈的状态

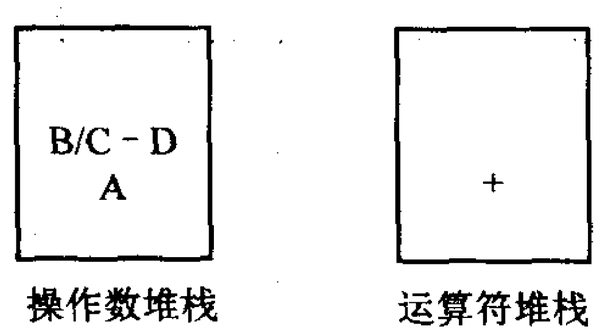


图 3-11 扫描结束后计算“-”时堆栈状态图

步骤 7：取出“+”和两个操作数，结果压入操作数栈，计算结束，如图 3-12 所示。

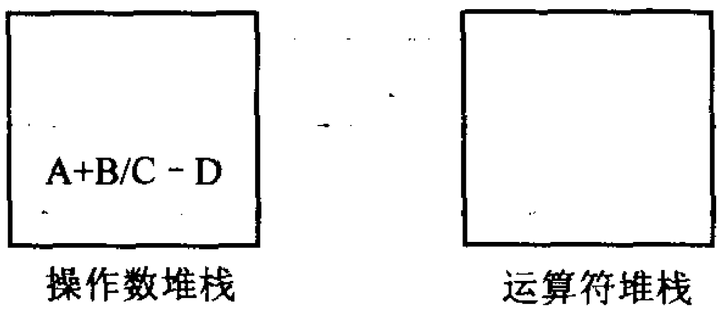


图 3-12 最后堆栈的状态

由以上的步骤可以看出，中缀表达式的计算需要使用两个堆栈，并且计算比较繁琐。而后缀(或前缀)表达式的计算只需要一个堆栈，建立一个操作数堆栈，将表达式从左向右扫描，只要是操作数，压入操作数堆栈；只要是运算符，从栈中取出元素计算，计算结果存入操作数堆栈，直到扫描表达式结束为止。由此可见，后缀表达式的处理比中缀表达式的处理简单的多。

值得注意的是，我们日常生活中习惯使用中缀表达式，而计算机使用后缀或前缀处理比较方便，所以对表达式的计算，就需要预先处理一下，最好是将中缀表达式转换为后缀(前缀)表达式。这就是我们考虑的关键问题：如何将中缀表达式转换为后缀(前缀)表达式。在中缀变后缀时，操作数的顺序不会发生变化，只有运算符的顺序可能发生变化，同时又没有括号，所以在转换的过程中，只要碰到操作数，可以直接输出，而碰到运算符和括号进行相应的处理即可。

转换原则如下：

(1) 从左至右读取一个中序表达式。

(2) 若读取的是操作数，则直接输出。

(3) 若读取的是运算符，分 3 种情况。

- 该运算符为左括号“(”，则直接存入堆栈。
- 该运算符为右括号“)”，则输出堆栈中的运算符，直到取出左括号为止。
- 该运算符为非括号运算符，则与堆栈顶端的运算符做优先权比较，若较堆栈顶端运算符高或相等，则直接存入堆栈；若较堆栈顶端运算符低，则输出堆栈中的运算符。

(4) 当表达式已经读取完成，而堆栈中尚有运算符时，则依次序取出运算符，直到堆栈为空，由此得到的结果就是中缀表达式转换成的后缀表达式。

假设栈顶元素的运算符是 θ_1，新读到的运算符是 θ_2，表达式的终止符是“#”，以“>”和“<”表示优先级的高与低。运算符优先级如表 3-2 所示。

表 3-2　运算符优先级关系表

θ_1 \ θ_2	+	−	×	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
×	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

根据此优先关系表和转换规则，可以实现将中缀表达式转换为后缀表达式，从而被计算机所接受。

例如：*A*/*B* − *C**(*D*+*E*)。

(1) 输出 *A*，“/”入栈；

(2) 输出 *B*；

(3) 读到“ − ”时，将“ − ”优先级和栈中的“/”优先级比较，小于，则输出“/”，然后将“ − ”入栈；

(4) 输出 *C*，将“*”优先级和“ − ”比较，大于，将“*”入栈；

(5) 将“(”入栈，输出 *D*，将“+”入栈，输出 E；

(6) 读到“)”时，输出“+”，一直到“(”出栈；

(7) 输出“*”，输出“ − ”，堆栈为空结束。

输出的结果 *AB*/*CDE*+*-就是转换完成的后缀表达式。

其转换算法为：

```
class Stackapp
{
    public static void main   (String args[])
    {
     StackArray Operator=new Stackarray();     //运算符堆栈
     String     inorder=new String();              //声明前序表达式字符串
     Int    inposition=0;                                 //前序表达式位置
     Int Operator1=0;       //运算符
     System.out.print("Please input the inorder expression： ");
     //读取前序表达式存入字符串
     ConsoleReader console=new ConsoleReader(System.in);
     Inorder=console.readline();
     System.out.print("The postorder expression is[");
```

```
 While  (true)
{
//判断是否为运算符
if  (Operator.Isoperator(inorder.charat(inposition)))
{
if  (operator.top==-1 ||(char)inorder.charat(inposition)=='(')
//将运算符存到堆栈中
operator.push(inorder.charat(inposition));
else
{
if((char)inorder.charat(inposition)==')')
{
//取出运算符直到取出'('
if(operator.astack[operator.top]!=40)
{
operator1=operator.pop();
System.out.print((char)operator1);
}
}
if (operator.priority(inorder.charat(inposition))
operator.priority(operator.astack[operator.top])!=-1)
operator1=operator.pop();
if(operatorl!=40)
system.out.print((char)operator1);
operator.push(inorder.charat(inposition));
system.out.print(inorder.charat(inposition));
if(inposition)>=inorder.length())
while(operator.top!=-1)  //取出在堆栈中所有的运算符
operator1=operator.pop();
system.out.print((char)operator1);
System.out.println(")");
int [] astack=new int [maxsize];
 //存入堆栈数据
public void push(int value)
if  (top>=maxsize)        //判断是否已超出堆栈最大容量
system.out.println("The stack is full!");
top++;
astack[top]=value;    //栈顶指针加 1，压入堆栈
  //从堆栈中取出数据
public int Pop()
{  int  temp;
if (Top<0)        //判断堆栈是否为空
{      system.out.println("The stack is empty!");
```

```
return –1;
}
temp=astack[top];          //将取出数据暂存于变量中
top--;                     //栈顶指针-1
return temp;
}
//判断是否为运算符
 public boolean isoperator(int operator)
 {
 if(operator==43   || operator==45
 ||operator==42   || operator==47
 ||operator==40   || operator==41)
 return true;       //+-*/运算符
 else
 return false;
 }
//判断运算符的优先权
public int priority(int operator)
 {
 //+-(运算符
  if (operator==43 ||operator==45 || operator==40)
  return 1;
  else if(operator==42 || operator==47)       //*/运算符
  return 2;
  else
  return   0;
 }
}
```

按照以上的算法，可以在计算机内首先完成表达式的转换，然后再计算表达式的值，这样计算机在计算时相对比较节省系统资源。

3. 递归

递归问题实际上是程序或函数重复调用自己，并传入不同的变量来执行一种程序。譬如一个人上楼梯，他现在在底层，如果想登上 100 层台阶，可以用一种方法：如果想上 100 层，只要登上 99 层，再上一层即可；如果想上 99 层，只要登上 98 层；依次类推，如果想上 2 层，只要登上 1 层；登上 1 层的方法可以直接实现。解决这个问题，只要处理 1 层的问题，其他的层次只要调用自己的同时，层数(参数)不断减少即可。在编写递归程序时，虽然是自己调用自己，但首先应该知道可以直接解决的问题，对于不可直接解决的问题，可以将它逐渐向可以解决问题的方向引进。例如参数的逐渐减少或增加，能够到达直接解决问题的这一步；否则就变成死循环，也就是计算机不能解决的问题。

递归程序编写虽然简单，但在时间和空间上往往是不节省的。

递归是一种比较好的程序设计方法，比较典型的范例是汉内塔、数学上的阶乘以及最大公因子等问题。下面仅以阶乘问题来说明递归。

阶乘定义为：

$$n!=\begin{cases}1 & n=0\\ n*(n-1)! & n>1\end{cases}$$

程序设计方法：

(1) 递归结束条件　当阶乘小于或等于 1 时，返回 1。

(2) 递归执行部分　当阶乘大于 1 时，返回 $n!=n*(n-1)!$。

算法为：

```
public class factor
   {
    public static void main (string ards[])
       {
          int number;
          int factorial;
          system.out.print("please enter a number");
          consolereader console=new consolereader(system.in);
          number=console.readint();
     facterial=factor(number);
     system.out.print(number+"!");
     system.out.print("="+factorial);
   }
  public static int factor(int n)
  {
       if (n<=1)
          return 1;
       else
           return n*factor(n-1);     //自己调用自己，但是参数逐渐减少，最后总能够达到 n
                                       的值等于 0 的时候
    }
}
```

4. 递归的非递归实现

在递归程序中，主要就是一个堆栈的变化过程，程序执行过程中，堆栈是由系统自动实现的，但是读者应该能够将递归的程序变为非递归的实现。

在非递归程序中，需要了解的是什么数据需要或什么时间压入堆栈，什么数据需要或在什么时候弹出堆栈。

例如上例中的阶乘问题，使用非递归实现，可以考虑实现将不同的 n 压入堆栈，每次减 1，最后能够实现 0 的阶乘的计算，然后返回，直到堆栈为空为止。

```
public class factor
    {
     public static void main (string ards[])
        {
          int number;
          int factorial;
          system.out.print("please enter a number");
          consolereader console=new consolereader(system.in);
          number=console.readint();
        top=null;
while (number!=0)
          {
            top=new link(number,top);
            number--;
          }
            factorial=1;
        while (top!=null)
          { object it=top.element(); //保留弹出的元素
             top=top.next();          //修改栈顶指针
factorial=it*factorial;
}
return factorial;
}
```

3.2 队　　列

3.2.1 队列定义及基本概念

1. 队列定义

队的操作是在两端进行，其中一端只能进行插入，该端称为队列的队尾，而另一端只能进行删除，该端称为队列的队首。队列在我们日常生活中经常碰到，例如，排队买东西，谁先来，谁先买，买完就走，谁后来，谁在队的最后面排队；再如，同一台打印机打印多份文件，可以在打印任务栏中发现，有多个打印任务在排队等待打印，而当前正在打印第一份文件。一般情况下，入队操作又称为队列的插入，出队操作又称为队列的删除。队列的运算规则是 FIFO(First In First Out)，或者叫做先进先出规则。队列的逻辑图如图 3-13 所示。

图 3-13 队列入出队的逻辑图

队列的入、出队操作分别具有入队和出队指针，通常以 f(front)表示队首指针，r(rear)表示队尾指针。

队列的存储具有顺序存储和链式存储两种。

注意：

队列在顺序存储时，经常出现“假溢出”现象，解决“假溢出”现象的方法有很多种，例如，设定队首指针不动，只要插入元素，在队列的末尾直接插入，只要删除元素，从队首的位置直接删除就可以了，队首的指针一直是确定的。但是这种方法通常需要造成大量的数据元素移动。所以解决“假溢出”比较好的方法是，将队列采用循环队列方式存储，也就是使得队列的队首和队尾指针循环起来。队列主要用于某种先后顺序处理的问题中，如操作系统的作业调度、打印任务的调度等。

2. 队列的基本运算

队列的基本运算通常和堆栈的基本运算类似，有以下 6 种：

- 置空队；//构造一个空队列。
- 判对空；//队空返回真，否则返回假。
- 判对满；//队满返回真，否则返回假，仅限于顺序存储结构。
- 入队；//队列非满时，从队尾插入元素。
- 出队；//队列非空时，从队首删除元素。
- 取队首元素；//返回队首元素，不修改队首指针。

3.2.2 顺序队列

队列的顺序存储结构称为顺序队列，顺序队列实际上是运算受限的顺序表。和顺序表一样，顺序队列也必须用一个向量空间来存放当前队列中的元素。由于队列的队头和队尾的位置是变化的，因而要设置两个指针 front 和 rear 分别指示队头元素和队尾元素在向量空间中的位置，它们的初值在队列初始化时可以置为 0。入队时将新元素插入 rear 所指的位置，然后将 rear 加 1。出队时，删去 front 所指的元素，然后将 front 加 1 并返回被删元素。由此可见，当头尾指针相等时队列为空。在非空队列里，头指针始终指向队头元素，而尾指针始终指向队尾元素的下一个位置。

队列同堆栈一样也有上溢和下溢现象。此外，队列中还存在“假溢出”现象。所谓“假溢出”是指在入队和出队操作中，头尾指针不断增加而不减小或只减小而不增加，致使被

删除元素的空间无法重新利用，最后造成队列中有空闲空间，但是不能够插入元素，也不能够删除元素的现象。因此，尽管队列中实际的元素个数远远小于向量空间的规模，但也可能由于尾指针已超越向量空间的上界而不能进行入队或出队操作。该现象称为假上溢。

解决假上溢现象的方法有很多种，如固定队首指针，一旦删除元素，需要移动所有元素后，修改队尾指针，这样又可以插入元素了，只有在不能插入元素时，队列才满，否则可以一直插入元素，这种方法虽然能够解决“假溢出”，但需要造成大量的数据元素的移动。现在解决“假溢出”比较好的解决方案是使用循环向量，存储在循环向量中的队列称为循环队列(Circular Quene)，如图 3-14 所示。

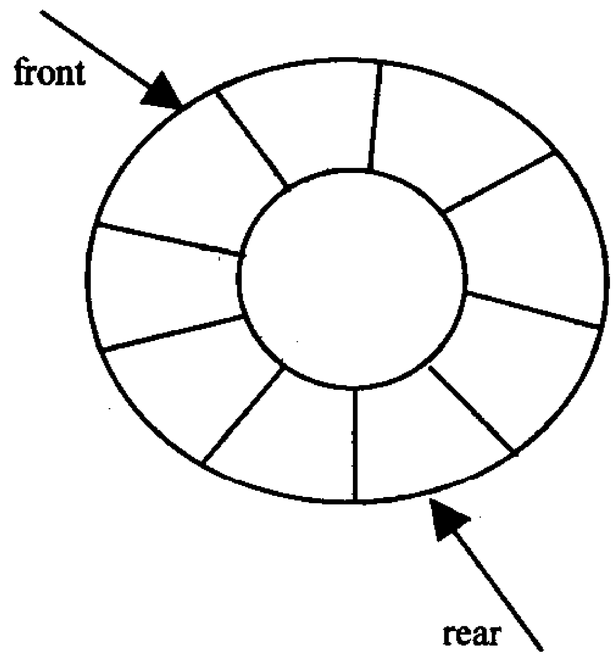


图 3-14　循环队列的逻辑图

假设向量的空间是 m，只要在入出队列时，将队首和队尾的指针对 m 做求模运算即可实现队首和队尾指针的循环，即队首和队尾指针的取值范围是 0 到 $m-1$ 之间。

入队时：rear=(rear+1)% maxsize

出队时：front=(front+1)% maxsize

但是入队和出队时 front 和 rear 的取值不能够确定队列的空和满。因为入队时，rear 指针不断增加一个，当 rear 指针“追上”front 指针时，队列已经满了，此时满的条件是 rear=front；反之，当出队时，front 指针不断增加一个，当 front 指针“追上”rear 指针时，队列已经空了，此时队列空的条件是 rear=front。由此可见，队空和队满的条件是相同的，都是 front=rear。

区分队空和队满的方法有多种。

方法一：可以设置浪费一个空间单元，也就是假定 rear 指向的是刚刚插入元素的位置，front 指向刚刚删除元素的位置。

也就是说在入队时，先不修改 rear 的值，而是先判断(rear+1) % maxsize=front，如果成立，表示队列已满(此时实际还有 front 指向的位置空闲)，出队时，只要判断 front=rear，如果成立表示队已空，否则只要 front=(front+1) % maxsize 直接删除元素即可。此种方法存储的数据元素个数是 maxsize - 1，是利用浪费一个空间来换取队空与满的条件。

方法二：设定一个变量来表示队列中的元素个数，利用该变量的值与队列的最大容量比较，如果该变量的值与最大容量相等，则表示队满；如果该变量的只为 0，则表示队列为空。此方法与第一种方法的不同之处在于，每次入队、出队一个元素时，需要修改队列

中的数据元素的个数。

以上两种方法，虽然都能够区分队空与队满，但是都需要增加一个存储单元的结构开销。

实现循环队列需要两个指针的开销，分别是 front 和 rear，需要定义数组的最大下标，在实现入/出队时，一定要遵循“先进先出”的规则。

循环队列的基本运算为：

```
class squeue{
private static final int defaultsize=10;
private int size;
private int front;
private int rear;
private object[] listarray;
squeue()
{setup(defaultsize);
}
squeue(int sz){setup(sz);}                    //设置大小
void setup(int sz)
{size=sz+1;
 fronr=rear=0;
listarray=new object[sz+1];}                  //队列初始化
public void clear()                           //队列清空
{front=rear=0;}
 public void enequeue(object it)              //入队一个元素
 { assert.notfalse(((rear+1)%size)!=front,"queueis full");
  rear=(rear+1)%size;
  listarray[rear]=it;}
  public object dequeue()                     //出队一个元素
 { assert.notfalse(!isempty(),"queue is empty");
    front=(front+1)%size;
    return listarray[front];
    }
 public object firstvalue()                   //取队首元素
 { assert.notfalse(!isempty(),"queue is empty");
   return listarray[(front+1)%size];
  }
public boolean isempty()                      //测试队列是否为空
{return front==rear;}
 }    //class Squeue
```

实现方法中，队首存放在数组中编号较低的位置(沿顺时针方向)，队尾存放在数组中编号较高的位置。

3.2.3　链式队列

定义链队列的存储结构基本和线性表的定义相同，不过需要在结构中指明的是队首和队尾的数据类型不再是整型而是指针类型。

队列的各种运算比链式存储的普通线性表运算实现时要方便得多，主要原因是队列的各种运算只能在队列的两端操作，队列是特殊的线性表，我们只要考虑对线性表的两端操作的情况，并且只能一端插入(队首)，另一端删除(队尾)；而线性表除此之外(不限定一端插入、一端删除)，还需要考虑中间结点的插入、删除、查询等操作，可以把队列的入出队运算当作线性表两端进行插入、删除的特例。

主要考虑队列的入队和出队算法。其原因是在队列的基本运算中有 6 种：判断队空、队列初始化、判断队列满(仅限于顺序存储的情况下)、入队元素、出队元素、取队首元素等。而入队时需要操作的步骤是初始化，然后判断队列是否为满，如果不满，则可以插入元素(入队)；出队时，需要考虑的操作步骤是判断队列是否为空，如果不空，删除元素(出队)，出队之前，保存队首元素。也就是说，队列的入出队运算包含了其他的 6 种基本运算，取队首元素的运算，只是不用修改队首的指针而已。

以单向链表存储队列为例，只要保留末尾结点指针(假设是队尾指针)即可，主要原因是如果保存队首和队尾指针，结构开销较大。由于在循环队列中，已知任意结点，可以找到所有结点，所以只要保留一个结点就可以了。如果已知结点的指针是头结点指针，此时入队、出队运算的时间复杂度为 $O(n)$(主要时间花费在查询结点上)；而如果已知结点的指针是末尾结点指针，此时不需要查询结点，直接进行入队和出队运算，时间复杂度为 $O(1)$，各种运算实现也比较方便。

```
class linkqueue {
    private link front;       //队首指针
    private link rear;        //队尾指针
    public linkqueue(){ setup(); }
    public linkqueue(int sz){setup(); }
    //初始化
    private void setup()
    {front=rear=null; }
    public void clear() {fron=rear=null; }    //清空队列
    //插入元素
    public void enqueue(object it){
    if (rear!=null){
    rear.setNext(new link(it,null));
    rear=rear.next();
}
else front=rear=new link(it, null);
}
 //删除元素
```

```
public object dequeue(){
 assert.notfalse(!isempty());
 object it=front.element();
 front=front.next();
 if(front==null)   rear=null;
 return it;
 }
 //取队首元素
 public object firstvalue()
  {assert.notfalse(!isempty());
   return front.element(); }
  //测试队列是否为空
  public boolean isempty()
  {return front==null; }
   }   //clases linkqueue
```

注意，在出队算法中，一般只需修改队头指针。但当原队中只有一个结点时，该结点既是队头也是队尾，故删去此结点时亦需修改尾指针，且删去此结点后队列变空。

3.2.4 队列的应用

在客户/服务关系中，当请求的速度超过服务所能提供的速度时，应该想到使用队列解决。例如车辆进入收费点时，需要排队等候，恰好是满足先进先出的规则，使用队列比较容易解决；在计算机中使用编辑软件编辑文字之后，连续发出多条打印命令，此时打印任务也处于排队状态，适宜使用队列解决。

1. 合并两个队列

假设有两个队列，要求将两个队列合并到一起，合并时交替使用两个队列中的元素，并把剩余队列中的元素添加在最后，将产生的新队列返回。

```
Public   static ArrayQueue   merged(ArrayQueue   q1, ArrayQueue q2)
{ ArrayQueue newQueue= new   ArrayQueue();
  while (!q1.IsEmpty() &&!q2.IsEmpty)
 {newQueue.enqueue(q1.dequeue());
  newQueue.enqueue(q2.dequeue());
 }
while (!q1.IsEmpty())
newQueue.enqueue(q1.dequeue());
while (!q2.IsEmpty())
newQueue.enqueue(q2.dequeue());
return newqueue;
}
```

2. 模拟客户服务系统

在客户/服务关系中，当遇到请求速度超过服务所能提供的速度时，将自然地想到用队列来进行模拟实现，例如，当汽车到达收费站时，在它能进入某个服务口之前它需要排队等待，刚好体现了队列先进先出的运算规则，可以考虑使用队列实现之；再如，使用计算机进行打印时，往往发出多个打印命令，要求打印机打印，此时打印任务中有多个打印在排队等候，对打印任务而言，刚好体现了队列先进先出的运算规则，可以考虑使用队列实现之；超市的入口和出口排队时，也可以考虑使用队列实现，因为它们满足先进先出规则等。在实际应用过程中需要使用队列的例子有很多。下面以打印作业为例说明模拟客户服务系统。

对打印机作业用它的 ID 号以及它的大小进行标识，对打印作业的有效模拟需要使用随机产生的数字输入，一个用来产生每台打印机的平均打印速度，一个用于为每台打印机的各打印作业产生打印速度，一个用于产生打印作业的到达间隙，该 Random 类是 java.util.random 类的扩展类。

```
public class random extends     java.util.random
{private double mean;
    private double standardDeviation;
    public Random(double mean)
{this.mean=mean;
this.standardDeviation=mean;
}
public Random(double mean,double standardDeviation)
{this.mean=mean;
this.standardDeviation=standardDeviation;
}
public double nextGaussian()
{double x=super.nextGaussian();    //x=normal(0.0,    1.0)
return x*standardDeviation + mean;
}
public double nextExponential()
{return    -mean*Math.log(1.0 - nextDouble());
}
public int intNextExponential()
{return (int)Math.ceil(nextExponential());
}
}
```

其中的 nextGaussian()方法将返回一个随机数，随机数按给定的平均数和标准偏差值正态分布，它调用并覆盖了 java.util.random 类中的匿名方法，该匿名方法返回的随机数按平均数为 0.0，而标准偏差值为 1.0 进行正态分布。方法 nextExponential()返回的随机数按给

定平均数进行指数分布，这是用于随机到达间隙时间的一个比较准确的分布，它也用于产生作业的大小，而该大小将用于确定服务所需的时间。

每个打印作业是以下类的一个实例。

```
public class Client
{public static final int MEAN_JOB_SIZE:100;
private static Random randomJobSize:new Random(MEAN_JOB_SIZE);
private static int nextld=0;
private int id, jobSize, tArrived, tBegan, tended;
private Server server;
public Client(int time)
{   id=++nextld;
jobSize=randomJobSize.intNextExponentiai();
//   tArrived=time;
print(id,time,jobSize);
public   double   getJobSize()
{   return   jobSize;
}
public   int   getWaitTime()
{   return   tBegan   -   tArrived;
}
public   int   getServiceTime()
{   return   tEnded   -   tBegan;
}
public   void   beginService(Server   server,   int   time)
{   this.server   =   server;
// tBegan   =   time;
printBegins(server,id,time);
}
public   void   endService(int   time)
{      //tEnded   =   time;
printEnds(server,id,time);
Server   =   null;
}
public   string   toString()
{   return   "#"   +   id   +   "("   +   (int)Math.round(jobSize) +   ")";
}
private   static   void   print(int   job,   int   time,   double   size)
{   System. out. println("job   #"   +   job   +   "   arrives   at   time   "   +   time
+   "   with "   +   (int)Math. round(size)   +   " pages.");
private   static   Void   printBegins(Server   server,   int   job,   int   time)
{   System.Out.println("Printer   "   +   server   +   "   begins   job   #"   +   job
+   " at   time   "   +   time   +   ".");
```

```
}
private static Void printEnds(Server server, int job, int time)
{ System.out.println("Printer " + server + " ends job #" + job
+ " at time " + time + ".");
}
}
```

随机数产生器 randomJobSize 产生指数分布的作业大小，平均页数为 100 页。它被声明为 static，这是由于一个实例就足以产生所有作业的大小。同样，static int nextid 用于所有作业的标识数。

构造函数使用 nextid 计数器来为作业设置 id 号，同时它还使用了 randomJobSize 产生器来设置作业的大小，然后它打印一行输出，宣布该作业已经到达。

方法 beginService()将打印机赋值给 server 引用，并输出一行，表明打印已经开始，同样，endService()方法先打印一行，宣布该打印作业结束，然后把空值赋值给 server 引用。

每个打印机是以下类的一个实例。

```
public class Server
{private static Random randomMeanServiceRate:new Random(1.00, 0.20);
private static char nextid='A';
private Random randomServiceRate;
private char id;
private double meanServiceRate,serviceRate;
prmvate Client client;
private int timeServiceEnds;
public Server()
{id =    (char)nextid++;
meanServiceRate = randomMeanServiceRate.nextGaussian();
randomServiceRate = new Random(meanServiceRate,0.10);
}
public Client getClient()
{return client;
}
public void beginServing(Client client,int time)
{this.client = client;
serviceRate = randomServiceRate.nextGaussian();
client.beginService(this,time);
int serviceTime = (int)Math.ceil(client.getJobSize() / serviceRate);
timeServiceEnds = time + serviceTime;
}
public void endServing(int time)
{client.endService(time);
this.client = null;
```

```
}
public int getTimeServiceEnds()
{return timeServiceEnds;
}
public boolean isFree()
{return client = null;
}
public String toString()
{int percentMeanServiceRate = (int)Math.round(100*meanServiceRate);
int    percentServicerate = (int)Math.round(100*serviceRate);
return id + "(" + percentMeanServiceRate + "%"    + percentServiceRate +"%)";
}
}
```

随机数产生器 randomMeanServiceRate 产生一个平均数为 100.0，而标准偏差为 20.0 的正态分布的打印速度随机值，为每台打印机产生一个 meanServiceRate 值。正如本次运行所示，它为 Printer A 产生的打印速度值为 89%，它为 Printer B 产生的打印速度值为 97%，它为 Printer C 产生的打印速度值为 106%，而为 Printer D 产生的打印速度值为 128%。同样，随机数产生器 randomServiceRate 为某个打印作业产生正态分布的速度值。在所显示的运行中，它为作业#1 所产生的速度为 84%，为作业#3 所产生的速度为 87%，为作业#5 所产生的速度为 92%，这些数值来源于平均数为 89%的正态分布(对 Printer A)，而标准偏差值设为 10%。

方法 beginServing()将要打印的客户作业赋值给 client 引用，并从产生器 randomServicerate 那里获得正态分布的 serviceRate 值，然后将 beginService 消息发送给客户打印作业，接下来，赋值语句 int serviceTime=(int)Math.ceil(client.getJobSize() / serviceRate); 完成打印作业所需的时间(秒数)，该值是用作业大小(页面数)除以打印速度(每秒多少页)来获得的，然后将该值加入到 Server 对象中的 timeServiceEnds 时间段中。

下面是 main 方法所在的类定义。

```
import schaums.dswj.Queue;
public class ClientServerSimulation
{private static final int NUMBER OF SERVERS = 4;
private static final double MEAN_INTERARRIVAL_TIME = 20.0;
private static final int DURATION = 100;
private static Server[] servers = new Server[NUMBER OF_SERVERS];
private static Queue clients = new ArrayQueue();
private static Random random = new Random(MEAN_INTERARRIVAL_TIME);
public static void main(String[] args)
{for(int i=0;i<NUMBER OF SERVERS;i++)
servers[i] = new Server();
int timeofNextArrival = random.intNextExponential();
```

```
for (int t=0; t<DURATION; t++)
{if(t == timeofNextArrival)
{clients.enqueue(new Client(t));
print(clients);
timeofNextArrival += random. intNextExponential();
}
for (int i=0;i<NUMBER OF SERVERS;i++)
if (servers[i].isFree())
{if(!clients.isEmpty())
{servers[i].beginServing((Client)clients.dequeue(),t);
print(clients);
}
}
else if(t == servers[i].getTimeServiceEnds())
servers[i].endServing(t);
}
private static void print(Queue queue)
{int size = queue. size();
if(size == 0) System.out.println("The queue is now empty.");
else
System.out.println(" The queue flow contains" + size + "job" + (size>1?"s:":":") + queue);
}
}
```

主循环在每个时间间隔均迭代一次，称这样的模拟实现为时间驱动模拟，相反，如果主循环是在某个事件发生时才迭代一次，就称其为事件驱动的模拟实现。事件可以是新作业的到达、服务的开始或服务的结束。事件驱动模拟程序通常更为简单，但要求所有的服务器具有相同的工作速度。

思考和练习

1. 基础知识题

(1) 向顺序堆栈插入新元素分为 3 步：第 1 步，进行_____判断，判断条件是_____；第 2 步修改_____；第 3 步把新元素赋给_____。同样从顺序堆栈删除元素分为 3 步：第 1 步，进行_____判断，判断条件是_____；第 2 步把_____值返回；第 3 步_____。

(2) 设有一个栈，元素依次进栈的顺序为 *A*、*B*、*C*、*D*、*E*。下列_____是不可能的出栈序列。

(a) *A*,*B*,*C*,*D*,*E*　　(b) *B*,*C*,*D*,*E*,*A*　　(c) *E*,*A*,*B*,*C*,*D*　　(d) *E*,*D*,*C*,*B*,*A*

(3) 何谓队列的上溢现象？一般使用什么方法解决？试简述之。

(4) 简述栈与队的不同之处。

(5) 简述顺序存储队列为何采用循环队列的存储方式。

(6) 循环链表与单向链表在处理方法上有何不同之处。

(7) 简述中缀、前缀、后缀的不同点。

(8) 简述一元多项式的栈运算过程。

(9) 设将整数 1、2、3、4 依次进栈，但只要出栈时栈非空，则可将出栈操作按任何次序压入其中，回答下述问题：

① 若入、出栈次序为 Push(1)、Pop()、Push(2)、Push(3)、Pop()、Pop()、Push(4)、Pop()，则出栈的数字序列是什么(这里 Push(i) 表示 i 进栈，Pop()表示出栈)?

② 能否得到出栈序列 1423 和 1432，并说明为什么不能得到或者如何得到。

③ 分析 1、2、3、4 的 24 种排列中，哪些序列是可以通过相应的入出栈操作得到的。

(10) 链栈中为何不设置头结点?

(11) 循环队列的优点是什么？如何判别它的空和满?

(12) 设长度为 n 的链队列用单循环链表表示，若只设头指针，则入队出队操作的时间是什么？若只设尾指针呢?

(13) 简述以下定义：栈、队列和递归。

(14) 从现实生活中举例说明栈和队列的特征。

(15) 举例说明栈的“上溢”、“下溢”现象。

(16) 举例说明顺序队列的“假溢出”现象。

(17) ①如果以链表作为栈的存储结构，则退栈操作时_____。

(a) 必须判别栈是否满　(b) 判别栈元素的类型

(c) 必须判别栈是否空　(d) 对栈不作任何判别

② 设 C 语言数组 Data[m+1]作为循环队列 SQ 的存储空间，front 为队头指针，rear 为队尾指针，则执行出队操作的语句为_____。

(a) front=front+1　(b) front=(front+1)%m

(c) front=(front+1)%(m+1)　(d) rear=(rear+1)%m

(18) 栈称为_____线性表。队称为_____线性表。设一个链栈的栈顶指针为 LS，栈中结点的格式为：

INFO	DATA

则栈空的条件是_____，如果栈不为空，则退栈操作步骤为_____。

(19) 某队列初始为空，若它的输入序列为 a、b、c、d，它的输出序列应为_____。

(a) a，b，c，d　(b) d，c，b，a　(c) a，c，b，d　(d) d，a，c，b

(20) 当 4 个元素的进栈序列给定以后，由这 4 个元素组成的可能的出栈序列应该有_____。

(a) 24 种　(b) 17 种　(c) 16 种　(d) 14 种

(21) 设 n 个元素的进栈序列为 1，2，3，……，n，出栈序列为 p_1，p_2，p_3，……，p_n，若 $p_1=n$，则 $p_i(1\leqslant i<n)$的值为_____。

(a) i　　(b) $n-i$　　(c) $n-i+1$　　(d) 有多种可能

(22) 设 n 个元素的进栈序列为 p_1，p_2，p_3，……，p_n，出栈序列为 1，2，3，……，n，若 $p_n=1$，则 p_i $(1\leqslant i<n)$的值为_____。

(a) i　　(b) $n-i$　　(c) $n-i+1$　　(d) 有多种可能

(23) 若堆栈采用顺序存储结构，正常情况下，往堆栈中插入一个元素，栈顶指针 top 的变化是_____。

(a) 不变　　(b) top=0　　(c) top--　　(d) top++

(24) 若堆栈采用顺序存储结构，正常情况下，删除堆栈中一个元素，栈顶指针 top 的变化是_____。

(a) 不变　　(b) top=0　　(c) top--　　(d) top++

(25) 若队列采用顺序存储结构，元素的排列顺序_____。

(a) 与元素的值的大小有关

(b) 由元素进入队列的先后顺序决定

(c) 与队头指针和队尾指针的取值有关

(d) 与作为顺序存储结构的数组的大小有关

(26) “链接队列”这一概念不涉及_____。

(a) 数据的存储结构　　(b) 数据的逻辑结构

(c) 对数据进行的操作　　(d) 链表的种类

(27) 在循环队列中，若 front 与 rear 分别表示队头元素和队尾元素的位置，则判断循环队列队空的条件是_____。

(a) front=rear+1　　(b) rear=front+1　　(c) front=rear　　(d) front=0

2. 算法设计题

(1) 用标志位方式设计在循环队列中入队算法。

(2) 写出顺序存储队列入队和出队算法。

(3) 写出顺序存储栈的入栈和出栈算法。

(4) 试写出利用两个堆栈 S_1、S_2 模拟一个队列的入、出队算法。

(5) 回文是指正读和反读均相同的字符序列，如 abba 和 abdba 均是回文，但 good 不是回文。试写一个算法判定给定的字符向量是否为回文。(提示：将一半字符入栈)。

(6) 利用栈的基本操作，写一个将栈 S 中所有结点均删去的算法。

(7) 利用栈的基本操作，写一个返回栈 S 中结点个数的算法。

(8) 设计算法判断一个算术表达式的圆括号是否正确配对。(提示：对表达式进行扫描，凡遇“(”就进栈，遇“)”就退掉，栈顶的“(”表达式被扫描完毕，栈应为空。

(9) 一个双向栈 S 是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈 S 设计初始化 InitStack(S)、入栈 Push(int i)和出栈 Pop(int i)等算法，

其中 i 为 0 或 1，用以指示栈号。

(10) 用第 2 种方法，即少用一个元素空间的方法来区别循环队列的队空和队满，试为其设计置空队、判队空、判队满、出队、入队及取队头元素等 6 个基本操作的算法。

(11) 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素站点(注意不设头指针)，试编写相应的置空队、判队空、入队和出队等算法。

(12) 对于循环向量中的循环队列，写出求队列长度的公式。

(13) 假设循环队列中只设 rear 和 quelen 来分别指示队尾元素的位置和队中元素的个数，试给出判别此循环队列的队满条件，并写出相应的入队和出队算法，要求出队时需返回队头元素。

(14) 对于循环队列，试写出求队列长度的算法。

(15) 某汽车轮渡口，过江渡船每次能载 10 辆车。过江车辆分为客车类和货车类，上渡船有如下规定：同类车先到先上船；客车先于货车上渡船，且每上 4 辆客车，才允许上一辆货车；若等待客车不足 4 辆，则以货车代替，若无货车等待允许客车都上船。试写一算法模拟渡口管理。

(16) 可以在一个数组中保存两个栈：一个栈以数组的第一个单元作为栈底，另一个栈以数组的最后一个单元作为栈底。设 *S* 是其中一个栈，试分别编写过程 push(S,X)(元素 X 入栈)、出栈 pop(S)和取栈顶元素 top(S)。提示：设其中一个栈为 0，另一个栈为 1。

(17) 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(注意不设头指针)，试编写相应的初始化队列、入队列和出队列算法。

(18) 假设以数组 cycque[m]存放循环队列的元素，同时设变量 rear 和 quelen 分别指示循环队列中队尾元素位置和内含元素的个数。试给出此循环队列的队满条件，并写出相应的入队列和出队列的算法。

第4章　数组和广义表

本章介绍多维数组的逻辑结构特征及其存储结构、特殊矩阵(主要是稀疏矩阵)的压缩存储及广义表的概念。

本章的学习目标：

- 多维数组的存储方式；
- 矩阵的压缩存储方式；
- 广义表的定义及其表头表尾的运算；
- 稀疏矩阵的存储表示。

4.1 多维数组

4.1.1 数组定义

数组是数据结构的基本结构形式，它是一种顺序式的结构。数组是存储同一类型数据的数据结构，使用数组时需要定义数组的大小和存储数据的数据类型。数组分为一维数组和多维数组。数组的维数是由数组下标的个数确定的，一个数组下标的数组称为一维数组，两个及其以上数组下标的数组称为多维数组。从这个意义上讲，确定了对于数组的一个下标总有一个相应的数值与之对应的关系；或者说数组是有限个同类型数据元素组成的序列。

数组的基本操作包括：

```
initarray(&A);          //初始化数组
destroyarray(&A);       //销毁数组
assign(&A,e);           //数组赋值
value(A,&e);            //取数组的某个元素
copyarray(M,&T);        //复制一个数组
printarray(M);          //打印数组的元素
```

1. 一维数组

一维数组是指下标的个数只有一个的数组，有时称为向量，是最基本的数据类型，在Java中需要事先声名，程序才能够在编译过程中预留内存空间。声明的格式一般是：

<数据类型> <变量名称> [ ]= new <数据类型> [<数组大小>];

例如：float numbera=new int[5];表示声明一个长度为 5 的浮点数据类型的一维数组，数组名称为 numbera。

2. 多维数组

多维数组是指下标的个数有两个或两个以上，我们比较常用的是二维数组(因为三维以上的数组存储可以简化为二维数组的存储)。下面以二维数组为例说明多维数组。二维数组的声明同一维数组。格式为：

<数据类型> <数组名称> [ ] [ ]=new <数据类型>[size1] [size2];

例如：int data[][]=new int [5][4];表示声明一个二维数组名称为 data 的整型数组，共有 5 行，每行有 4 列，共可存储 20 个数据元素。

4.1.2　数组的存储

1. 一维数组的存储

一维数组的数据存储按照顺序存储，逻辑地址和物理地址都是连续的。如果已知第一个数据元素的地址 $loc(a_1)$，则第 i 个元素的地址 $loc(a_i)$为：

$$loc(a_i)=loc(a_1)+(i-1)*c$$

假设数组的下标从 1 开始，只要求出第 i 个元素之前存放了多少个数据元素即可(实际上有 $i-1$ 个元素)，每个元素占有 c 个存储单元，再乘以 c，就是第 i 个元素的起始地址。

如果下标从 0 开始，则第 i 个元素之前就有 i 个元素，此时上面的公式就变为：

$$loc(a_i)=loc(a_1)+i*c$$

由此可见，求数组中数据元素的地址，已知条件必须是知道第一个元素的地址，然后主要是找出该元素之前已经存储了多少个数据元素。在一维数组中，只要知道任何一个元素的地址即可求出其他元素的地址，但在多维数组中，已知条件必须是第一个数据元素地址。

2. 多维数组

以二维数组的顺序存储为例说明，二维数组在顺序存储时一般有两种。

(1) 行优先顺序：存储时先按行从小到大的顺序存储，在每一行中按列号从小到大存储。

(2) 列优先顺序：存储时先按列从小到大的顺序存储，在每一列中按行号从小到大存储。

以上的两种存储顺序中，第一个被存放的元素总是第一行第一列的数据元素，所以该元素的地址是我们的已知条件。

同样在二维数组中比较典型的是计算数据的存储位置。

假设二维数组是 $m*n$ 的二维数组(共有 m 行，每行有 n 列)。第一个数据元素的地址是 $loc(a_{11})$，则第 i 行第 j 列的数据元素的地址的计算公式应为(按照行优先顺序存储)：

$$loc(a_{ij})=loc(a_{11})+[(i-1)*n+j-1]*c$$

假设下标从 1 开始，我们需要计算出 i 行前面已经存储了 $i-1$ 行元素，每行有 n 个元素，共有 $(i-1)*n$ 个数据元素，在第 i 行元素中，j 列之前有 $j-1$ 个数据元素，共有 $(i-1)*n+j-1$ 个元素，每个元素占有 c 个存储单元，只要乘以 c 就可以列出地址。其中 $loc(a_{ij})$ 表示第 i 行第 j 列数据元素的内存的起始位置，$loc(a_{11})$ 表示第一个数据元素的内存位置，c 表示每个数据元素所占有的内存空间的大小，如果下标从 0 开始，只要不用减 1 即可。

如果按列优先顺序存储，则地址的计算为：

$$loc(a_{ij})=loc(a_{11})+[(j-1)*m+i-1]*c$$

假设下标从 1 开始，其中 $loc(a_{ij})$ 表示第 i 行第 j 列的数据元素的内存起始位置，$loc(a_{11})$ 表示第一个数据元素的内存位置，c 表示每个数据元素所占有的内存空间的大小。主要还是计算第 i 行第 j 列元素之前有多少个数据元素。如果下标从 0 开始，只要不用减 1 即可。

按此公式可以推广到多维数组的数据元素的地址计算(假设按照行优先顺序存储)：

m 行 n 列纵标为 k 的三维数组，假设第一个元素的地址是 $loc(a_{111})$，如果按行优先顺序存储，i 行 j 列纵标为 p 的数据元素的地址为(可以将它分解为二维数组)：

$$loc(a_{ijp})=loc(a_{111})+[(i-1)*n*k+(j-1)*k+p-1]*c$$

如果下标从 0 开始，只要不用减 1 即可。

读者可以从以上的地址公式中找出一定的地址计算规律：多维数组中按行优先计算公式用一个下标乘以后面的最大值。

例如四维数组 $m*n*k*l$ 中，地址的公式为：

$$loc(a_{ijpq})=loc(a_{1111})+[(i-1)*n*k*l+(j-1)*k*l+(p-1)*l+q-1]*c$$

如果下标从 0 开始，只要不用减 1 即可。

下面以二维数组为例说明多维数组的基本操作。

4.1.3　显示二维数组的内容

假设二维数组以行为主序存储，将它转换为按列为主序的存储算法。

```
public class arrayzhuan
{
   pubilc static void main (string args[ ])
   {int [ ] [ ] data={{9,7,6,6},{3,5,3,3},{6,6,4,7},{7,5,1,4},{1,2,8,0}};      //定义二维数组
int   rowdata[]=new int[20]          //用来存储以行为主序的上述二维数组
```

```
int   coldata[]= new int [20]          //用来存储以列为主序的上述二维数组
int i,j;
system.out.println("输出二维数组");
      for (i=0;i<5;i++)
      {
       for (j=0;j<4;j++)
       system.out.print(" "+data[i][j]+" ");
       system.Out.println("");
       }
       for   (i=0;i<5;i++)   //转换为以行为主序的一维数组
        for (j=0;j<4;j++)
        rowdata[i*4+j]=Data[i][j];
        system.out.println("");
        system.out.println("the rowmajor matrix:");
       for (i=0;i<20;i++)                  //输出以行为主序的数组的值
        system.Out.print(" "+rowdata[i]+" ");
       system.out.println("");
      for (i=0;i<5;i++)       //转换以列为主序的一维数组
      for (j=0;j<4;j++)
       coldata[j*5+i]=data[i][j];
          system.out.println("");
          for (i=0;i<20;i++)            //输出以列为主序的数组的值
             system.out.print(" "+Coldata[i]+" ");
          system.out.println("");
        }
    }
```

一般情况下，只要定义了数组的存储顺序，数组的存储顺序就不会改变了，所以对数组的各种操作后，应按照数组的已定义的存储顺序存储。也就是说，如果是按行优先顺序存储，在对数组操作后，即使改变了存储顺序，也应改变按照行优先顺序存储。

4.2　矩阵的存储

4.2.1　矩阵的压缩存储

所谓矩阵的压缩存储，也就是在存储数组时，尽量减少存储空间，但是数组中的每个元素必须存储，所以在矩阵存储中，如果有规律可寻，只要存储其中一部分，而另一部分的存储地址可以通过相应的算法将它计算出来，从而占有比较少的存储空间达到存储整个矩阵的目的。

矩阵的压缩存储仅是针对特殊矩阵的，而对于没有规律可循的二维数组则不能够使用矩阵压缩存储。

二维数组(矩阵)的压缩存储一般有 3 种，它们分别是对称矩阵、稀疏矩阵和三角矩阵。3 种矩阵中以稀疏矩阵比较常见。

1. 对称矩阵

若 n 阶矩阵 A 中的元素满足以下条件：

$$a_{ij}=a_{ji} \qquad i\geqslant 1,\ j\geqslant 1$$

则称为 n 阶对称矩阵。

对于对称矩阵，如果不采用压缩存储，占有的存储单元有 n^2 个，因为是对称矩阵，所以只要存储对角的数据元素和一半的数据元素即可，占有的存储单元有 $n(n-1)/2$ 个。如果以行序为主序存储其下三角(包括对角线)的元素，其上三角的元素可以推算出来。

如果用一维数组存储一个对称矩阵，只要将对称矩阵存储在一个最大下标为 $n(n-1)/2$ 的一维数组 S 中即可。此时按照行优先顺序存储，数据元素 a_{ij} 与数组 S 的下标 k 的对应关系为：

$$k=\begin{cases} i(i-1)/2+j & \text{当 } i\geqslant j \text{ 时} \\ j(j-1)/2+i & \text{当 } i<j \text{ 时} \end{cases}$$

对于任意给定的一组下标(i,j)，均可在 S 中找到元素 a_{ij}，反之，对所有元素都能够确定在 S 中位置，当 $i<j$ 时，根据对称矩阵的性质推算即可。由此可以看出，对称矩阵的存储可以使用一维数组 S，占用的空间不再是 n^2，而是 $n(n-1)/2$，空间减少了接近一半，实现了二维数组的压缩存储。

所谓对角矩阵，也就是指矩阵的所有非零元素都集中在以主对角线为中心的带状区域中，即除了主对角线上和直接在主对角线上、下方若干条对角线上的元素之外，其余元素皆为零。

下面给出一个对角矩阵的例子(三对角矩阵)：

$$B=\begin{bmatrix} b_{11} & b_{12} & & & & \\ b_{21} & b_{22} & b_{23} & & & \\ & b_{32} & b_{33} & b_{34} & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-1\,n-2} & b_{n-1\,n-1} & b_{n-1\,n} \\ & & & & b_{n\,n-1} & b_{n\,n} \end{bmatrix}$$

也可以按照某个原则(或者以行序为主序，或者以列序为主序，或者按对角线的顺序)

将对角矩阵 *B* 的所有非零元素压缩存储到一个一维数组 LTB[1……3*n*-2]中。这里不妨仍然以行序为主序的原则对 *B* 进行压缩存储，当 *B* 中任一非零元素 b_{ij} 与 LTB[*k*]之间存在着如下一一对应关系：

$$k=2*i+j-2$$

时，则有 b_{ij}=LTB[*k*]。称 LTB[1……3*n* - 2]为对角矩阵 *B* 的压缩存储。

上面讨论的几种特殊矩阵中，非零元素的分布都具有明显的规律，因而都可以被压缩存储到一个一维数组中，并能够确定这些矩阵的每个非零元素在一维数组中的存储位置。但是，对于那些非零元素在矩阵中的分布没有规律的特殊矩阵(如稀疏矩阵)，则需要寻求其他方法来解决压缩存储问题。

2. 稀疏矩阵

对稀疏矩阵很难下一个确切的定义，它只是一个凭人们的直觉来理解的概念。一般认为，一个较大的矩阵中，零元素的个数相对于整个矩阵元素的总个数所占比例较大时，该矩阵就是一个稀疏矩阵。例如，有一个 6×6 阶的矩阵 *A*，其 36 个元素中只有 8 个非零元素，那么，可以称矩阵 *A* 为稀疏矩阵。

稀疏矩阵一般是指矩阵中的大部分元素为零，仅有少量元素非零的矩阵；或者说矩阵 $A(m\times n)$中有 *S* 个非零元素，如果 *S* 远远小于矩阵的元素总数，则称 *A* 为稀疏矩阵。稀疏矩阵的存储一般只要保存非零元素即可，对于零元素可以不予保存，这样就可以实现稀疏矩阵的压缩存储。

稀疏矩阵的压缩存储采用三元组的方法实现。其存储规则是每一个非零元素占有一行，每行中包含非零元素所在的行号、列号、非零元素的数值。为完整描述稀疏矩阵，一般在第一行描述矩阵的行数、列数和非零元素的个数。其逻辑描述为：

(row　col　value)

其中 row 表示行号，col 表示列号，value 表示非零元素的值。

如果每个非零元素按照此种方法存储，虽然能够完整地描述非零元素，但如果矩阵中有整行(或整列)中没有非零元素，此时可能不能够还原成原来的矩阵。所以为了完整地描述稀疏矩阵，在以上描述的情况下，如果增加一行的内容，该行包括矩阵的总的行数、矩阵的总的列数、矩阵中非零元素的个数，就可以还原为原来的矩阵描述了。

例如，稀疏矩阵为：

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

该矩阵是一个 5×6 阶矩阵，如果一个元素占有 1 个存储单元，应该占有 30 个存储单

元，如果使用三元组存储，则应该写为：

$$\begin{pmatrix} 5 & 6 & 4 \\ 2 & 2 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 4 \\ 4 & 3 & 9 \end{pmatrix}$$

第一行按列号依次描述的是：矩阵共有 5 行 6 列 4 个非零元素。

从第二行开始每列依次描述的含义是：非零元素所在的行号、列号和非零元素的值。

所占的空间是 5×3=15 个存储单元。

归纳起来，若一个稀疏矩阵有 t 个非零元素，则需要用 t+1 行的三元组来表示稀疏矩阵。到底矩阵何时使用三元组存储呢？一般对 $m \times n$ 的矩阵来说，只要满足$(t+1)*3 \leqslant m*n$这个条件，使用三元组存储可以节省空间，否则更加浪费空间，也就没有必要使用三元组存储，所以稀疏矩阵中的非零元素的个数 t 是能否使用三元组存储的关键。

4.2.2　稀疏矩阵转换为三元组存储

矩阵的存储规则一般有两种：按行优先顺序存储和按列优先顺序存储。所谓按行优先顺序存储是指在存储过程中，按照行号从小到大存储，行号相同时，按照列号从小到大顺序存储。所谓按列优先顺序存储是指在存储过程中，按照列号从小到大存储，列号相同时，按照行号从小到大顺序存储。一般我们选择按行号优先顺序存储的较多。通常在确定了存储规则之后，无论对矩阵进行什么样的操作，规则应该不变。也就是说，当矩阵按行号优先顺序存储时，如果将矩阵转置后，变成按列优先顺序存储，此时仍然需要将此矩阵转换为按行号优先顺序存储。

首先应该将稀疏矩阵转换为三元组存储，然后再利用三元组的存储，实现对矩阵的各种运算。

将一个具体的稀疏矩阵转换为三元组存储的算法为：

```
public class arrayl
{
    public static void main (string   args[])
    {
      int [][]data={      {0   0   0   0   0   0},
                          {0   3   0   0   0   0},
                          {0   4   0   0   0   0},
                          {0   0   9   0   0   0},
                          {0   0   0   0   0   0}};   //假设一个 5×6 阶矩阵
        int comparessdata[][]=new int [10][3];   //假设一个 10×3 的三元组空间
        int index;                               //三元组的行号
        int i,j;
```

```
        index=0;
        for (i=0;i<5;i++)                    //输出矩阵中的所有元素
           {
            for (j=0;j<6;j++)
              system.out.print(" "+data[i][j]+ "    ");
              system.out.println("");
                }
for (i=0;i<5;i++)
   for (j=0;j<6;j++)
      if data[i][j]!=0
        { index++;
           compressdata[index][0]=i;              //非零元素的行号
           compressdata[index][1]=j;              //非零元素的列号
           compressdata[index][2]=data[i][j];    //非零元素的值
          }
    compressdata[0][0]=5;                         //给三元组的第 0 行赋值
    compressdata[0][1]=6;
    compressdata[0][2]=index;
    for (i=0;i<=index;i++)                        //输出三元组的所有元素
        {
        for (j=0;j<3;j++)
          system.out.print(""+compressdata[i][j]+ " ");
          system.out.println("");
        }
      }
}
```

对于矩阵的运算一般有矩阵的转置，在转置时值得注意的是：在矩阵的存储规则已经确定的情况下(如按行优先存储)，实现矩阵的运算(如转置)时，应仍然保留原来的存储规则。

假设 compressdata[][]中存储着一个行优先的三元组，共有 5 行 6 列，index 个非零元素，将它转置后存储在 compressdata1[][]中，并仍然以三元组存储并按行优先存储。

```
public class transarrayl
 {
public static void main (string    args[])
 {
 int compressdata1[][]=new int [10][3];   // 假设一个 10×3 的三元组空间
        int index;                          //三元组的行号
        int i,j;
        compressdata1[0][0]=compressdata1[0][1];          //给三元组的第 0 行赋值
        compressdata1[0][1]= compressdata1[0][0];
        compressdata1[0][2]=index;
        int n=compressdata[0][1];
```

```
int m=1;
    for (i=1;i<=n;i++)                    //检查所有非零元素的列号并实现转置
      {
      for (j=1;j<=index;j++)
        {
          if (compressdata[j][1]==i)
            {compressdata1[m][0]=compressdata[j][1];
             compressdata1[m][1]=compressdata[j][0];
             compressdata1[m][2]=compressdata[j][2];
             m++;   }
      for (i=0;i<=index;i++)          //打印转置后的三元组
      {
      for (j=0;j<3;j++)
       system.out.print(""+compressdata1[i][j]+ " ");
          system.out.println("");
        }
    }
}
```

打印的结果仍然是按行优先顺序存储，但这种转置方法有很多的重复运算，例如对已经转置过的非零元素，仍需要扫描，对没有非零元素的行也需要扫描，其时间复杂度是 $O(n*t)$，可见算法的效率不高。为了提高算法的效率，可以对该算法加以改进，实现三元组的快速转置。

改进的转置方法可以利用对原始的三元组的元素的扫描，直接确定该元素在转置后的三元组中的行，这样可以将原始三元组中的元素直接放在转置后的三元组中即可。这种方法需要增加两个一维数组的结构开销。

假设两个数组 num[]和 pos[]，其中 num[i]表示在 compressdata[][]中列号为 i 的非零元素个数，pos[i]表示在 compressdata[][]中列号为 i 的第一个非零元素转置后放在 compressdata1[][]中的行号。规定：

$$pos[1]=1$$

此时不难看出 pos[i]之间存在以下的关系：

$$pos[i]=pos[i-1]+num[i-1] \qquad i\geqslant 2$$

在整个算法的实现中，计算出每一个 pos[i]，然后对 compressdata[][]中所有元素扫描，直接将数据元素放在 compressdata1[][]中即可。

```
public class transarray2
    {
    public static void main (string    args[])
     {
```

```
int comparessdata1[][]=new int [10][3];   //假设一个 10×3 的三元组空间
        int pos=new int[10];                  //假设有 10 个非零元素
        int num=new int[10];
        int index;                            //三元组的行号
        int i,j;
        Compressdata1[0][0]=compressdata1[0][1];          //给三元组的第 0 行赋值
Compressdata1[0][1]= compressdata1[0][0];
Compressdata1[0][2]=index;
int n=compressdata[0][1];
int m=1;
     for (j=0;j<n;j++)                        //给 num[]赋初值
             { num[i]=0;}
     pos[1]=1;
    for (i=1;i<=index;i++)                    //计算 num[]的值
       {num[compressdata[i][1]]++;}
    for (i=2;i<=n;i++)                        //计算 pos[]的值
      pos[i]=pos[i-1]+num[i-1];
  for (i=1;i<=index;i++)                      //直接转置
  {
     compressdata1[pos[compressdata[i][1]][0]=compressdata[i][1];
      compressdata1[pos[compressdata[i][1]][1]=compressdata[i][0];
      compressdata1[pos[compressdata[i][1]][2]=compressdata[i][2];
      pos[compressdata[i][1]]++;   }
    for (i=0;i<=index;i++)            //打印转置后的三元组
      {
      for (j=0;j<3;j++)
       system.out.print(""+compressdata1[i][j]+ " ");
           system.out.println("");
         }
     }
}
```

快速转置算法的时间复杂度为 $O(n+t)$。相对前面的转置算法，算法的效率明显提高了很多，但同时增加了空间的开销。

稀疏矩阵的基本运算有很多，例如矩阵的乘法等，在此就不再叙述了，请读者自己实现。

值得注意的是：向量(Vector)就是数字的一个有穷序列，从几何上看，可以把二维向量(x,y)看成平面中的一个点；而三维向量(x,y,z)表示三维空间中的一个点。在线性代数中，可以经常见到 n 个元素的向量$(x_1,x_2,x_3,\cdots,x_n)$，并使用下标来表示某个元素。

在 Java 中，除以下两点以外，向量与数组完全相同：

- 一个向量是类 java.util.Vector 的实例
- 一个向量的长度可以改变。

4.3 广 义 表

4.3.1 广义表的定义

广义表是线性表的扩展，具体定义为 $n(n\geq 0)$个元素的有限集合。其中元素有以下两种类型：

- 一个原子元素(指不可再分的元素)
- 一个可以再分的元素(或称为一个子表)

如果所有元素都是原子元素，则称为线性表，如果含有子表，则是广义表。n 的值是广义表的长度，如果 n=0，称广义表为空表。

广义表的基本操作为：

```
initGlist(&L)                    //创建空的广义表
creatGlist(&L,S)                 //由 S 创建广义表 L
destroyGlist(&L)                 //销毁广义表 L
Glistlength(L)                   //求广义表的长度
Glistdepth(L)                    //求广义表的深度
Gethead(L)                       //求广义表 L 的头
Gettail(L)                       //求广义表的表尾
Insertfirst_Glist(&L,e)          //插入元素 e 作为广义表 L 的第一个元素
Deletefirst_Glist(&L,&e)         //删除广义表 L 的第一个元素，并用 e 返回其值
```

广义表一般记作：

$$LS=(a_1,a_2,\ \cdots\cdots,a_n)$$

其中 LS 是广义表的名称，n 是广义表的长度。

常见的广义表为：

A=()

B=(())

C=(a,b)

D=(A,B,C)

E=(a,E)

广义表中含有元素的个数称为广义表的长度，广义表中含有的括号对数称为广义表的深度。

从上述定义和例子可推出列表的 3 个重要结论：

(1) 列表的元素可以是子表，而子表的元素还可以是子子表……由此，列表是一个多层次的结构，可以用图形象地表示。例如图 4-1 表示的是列表 D。其中以圆圈表示列表，

以方块表示原子元素。

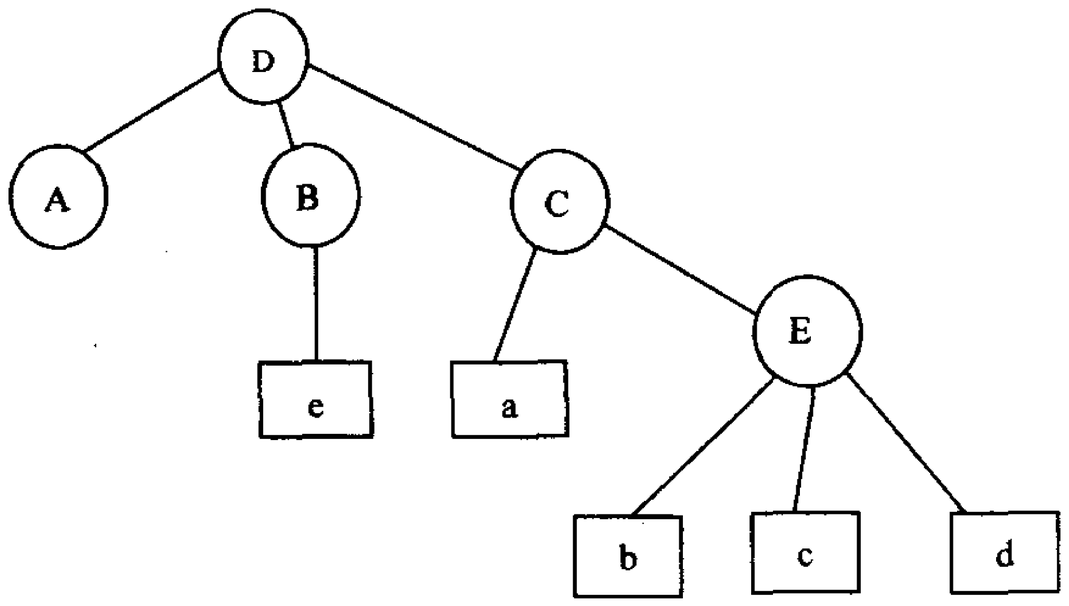


图 4-1　列表的图形表示

(2) 列表可为其他列表所共享。例如在上述例子中，列表 *A*、*B* 和 *C* 为 *D* 的子表，则在 D 中可以不必列出子表的值，而是通过子表的名称来引用。

(3) 列表可以是一个递归的表，即列表也可以是其本身的一个子表。例如，列表 *E* 就是一个递归的表。

广义表的表头是广义表中的第一个元素，而表尾则是去掉表头之后的所有元素，在广义表中通常利用求表头和表尾运算求得广义表中某个元素的值。

4.3.2　广义表的存储

广义表的存储方法有很多种，一般采用链表存储。采用链表存储时的结点存储的逻辑结构如图 4-2 所示。

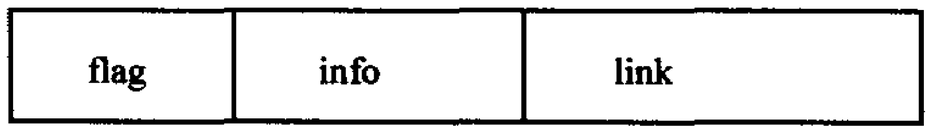


图 4-2　广义表的逻辑结构图

其中 flag 表示标志位，当 flag 为 0 时，该结点表示原子元素，当 flag 为 1 时，该结点表示子表；当 flag 为 0 时，info 表示原子元素的值，当 flag 为 1 时，info 表示指针，指向该子表的第一个结点；link 表示指针，指向广义表的下一个元素。

例如广义表 *A*=(*a*,(*b*,(*c*)),(*d*,*e*),*f*)，利用链表存储，其逻辑图如图 4-3 所示。

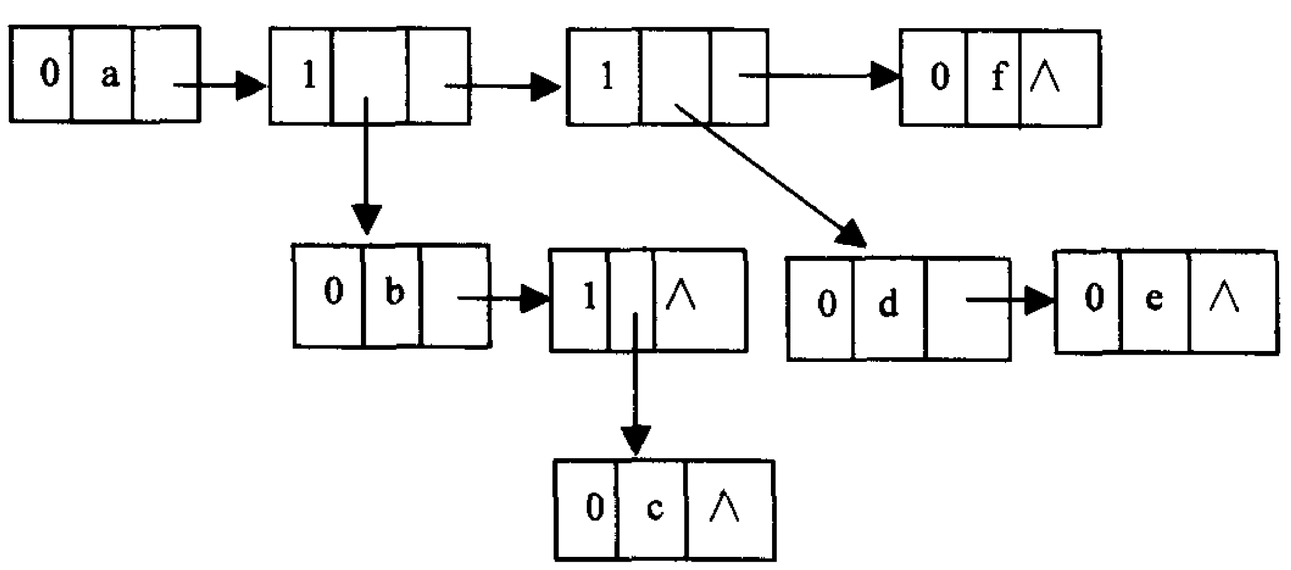


图 4-3　广义表的链式存储结构图

广义表可以采用多种方式实现，最简单的方法是使用数组实现，这对具有定长元素的链可以工作得很好，而对具有变长元素的链，可以把子表看成变长元素，如果要使用这种方法，需要每个子表的一些开始和结束标识。因为纯表等价于树，对于纯表可以使用链接分配的方法支持对表的子女的访问，需要附加标识符用来区别结点是原子还是子表。另一种方法是使用两个存储指针字段的链结点表示所有元素，除了原子以外，它只包含数据，指针可能包含一个标记位，来标识它指向什么，被指向的对象也可能存储一个标记位，来标识自己，标记把原子和表结点区分开，这种实现可以方便地支持可重入表和循环表，因为一个结点的指针可能指向任何其他结点。

思考和练习

1. 基础知识题

(1) 是否可以将多个不同数据类型的数据存储于同一个数组中？

(2) 如果声明一个大小为 20 的整数数组，是否一定要在数组中存满 20 个元素？

(3) 以行为主序和以列为主序说明数组的表示法。

(4) 若使用以行为主序的数组表示法是否比以列为主序的数组表示法要节省空间？

(5) 判断：在 Java 语言中声明一个数组为 Int Data[]=new int[20];则其可使用的空间为 Datat[1]～Data[20]。

(6) 下列说法哪个是不正确的？

(a) 每一个数组，皆有一个下标和一个元素值

(b) 下标是用来方便存取数据

(c) 元素值正是被存储数据的位置

(d) Java 中的数组下标从 0 开始

(7) 下列哪一个是声明一个大小为 30 的字符数组的正确方法？

(a) char Data[29]

(b) char Data[30]

(c) char Data[31]

(d) 以上皆非

(8) 若有一个整数数组 x 是采用以行为主序的表示法，已知其 x[3,5]的地址为 1000，x[5,7]的地址为 1200，则下列说法哪一个是不正确的？

(a) 行的个数为 49

(b) x[8,10]的地址为 1600

(c) x[3,8]的地址为 1016

(d) 整数数组每一个元素占 2 个字节

(9) 在一个长度为 n，包含 m 个原子元素的广义表中，_____。

(a) m 和 n 相等　　(b) m 不大于 n　　(c) m 不小于 n　　(d) m 与 n 无关

(10) 广义表的元素分为_____。

(a) 原子元素　　(b) 表元素　　(c) 原子元素和表元素　　(d) 任意元素

(11) 广义表 A=((),(a),(b,(c,d)))的长度为_____。

(a) 2　　(b) 3　　(c) 4　　(d) 5

2. 应用题

(1) 试说明何谓“静态内存配置”，其优、缺点各是什么。

(2) 若声明一个浮点数数组如下：

```
float Average[]=new float[30];
```

假设读数组的内存起始位置为 200，试求 Average[15]和 Average[27]的内存地址。

(3) 试利用 Java 语言写出在数组中插入元素和删除元素的子程序。

(4) 试说明何谓以行为主序和以列为主序。

(5) 试利用以行为主序的方法来说明数组在内存中的存储方式。

$$\begin{pmatrix} 9 & 7 & 5 & 0 & 1 \\ 3 & 5 & 4 & 6 & 8 \\ 1 & 8 & 2 & 5 & 4 \\ 3 & 0 & 7 & 1 & 8 \\ 9 & 5 & 1 & 8 & 6 \end{pmatrix}$$

(6) 若有一个二维数组 Data，排列方式为：Data[3][5]的内存地址为 3000，Data[4][6]的内存地址为 3600，试求 Data[5][7]的内存地址。

(7) 假设一个浮点数二维数组共有 7 行 9 列，数据存储方式采用以行为主序，且在内存上的起始地址是 300，试求出数组中第 6 行第 5 列的元素在内存中的地址。

(8) 试求出稀疏数组

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 3 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

压缩后的数组内容。

(9) 若有一个大小为 5×5 的上三角数组，试求出数组[3,4]这个元素在以行为主序和以列为主序两种方式转换后的一维数组的下标。

(10) 给定稀疏矩阵的存储表示后，实现以下操作：

- 在给定位置向矩阵插入一个元素
- 从矩阵的给定位置删除一个元素
- 检索矩阵中给定位置的元素
- 对一个矩阵进行转置
- 两个矩阵相加

第5章　树

树(tree)型结构是一类重要的非线性结构。树型结构反映了数据元素之间的层次关系和分支关系，非常类似于自然界中的树。树型结构在现实生活中广泛存在，如企业的组织结构图等。另外，在计算机科学中亦具有广泛的应用，如编译程序中源程序的语法结构就是用树型结构来表示的；数据库系统中也采用树型结构组织信息。

本章的学习目标：

- 树以及二叉树的概念、性质、存储结构和遍历算法；
- 一般树和二叉树的转换关系；
- 线索二叉树、哈夫曼树等典型树型结构的应用；
- 通过二叉树的定义了解数据之间的层次关系；
- 在掌握二叉树的遍历方法后，掌握线索二叉树；
- 了解哈夫曼树的应用和哈夫曼编码。

5.1　树 的 概 念

本节主要讲述树的定义；树的常用表示方式，即树型表示法、文氏图表示法、凹入图表示法以及广义表表示法；度、路径、结点的层数等树的基本术语。

5.1.1　树的定义

树是一种数据结构，表示为 *TREE*=(*D*,*R*);。

其中，*D* 是具有相同特性的数据元素的集合；*R* 是元素集合 *D* 上的关系集合，如果 *D* 中只含有一个数据元素，则 *R* 为空集。

或者用递归定义为：树是 *N*(*N*>0)个结点的有限集合。其惟一关系具有下列属性：集合中存在惟一的一个结点，称为树根，该结点没有前趋；除根结点外，其余结点分为 *M*(*M*≥0)个互不相交的集合，其中每一个集合都是一棵树，并称其为根的子树。

树的上述定义是一个递归定义，其说明了树的固有特性，即一棵树是由若干棵子树构成的，而子树又可由若干棵更小的子树构成，如图 5-1 所示。树中的结点一般没有次序之分，其次序可以任意颠倒。

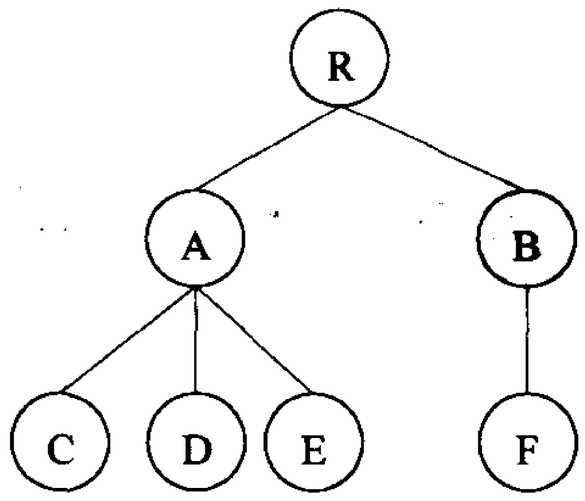
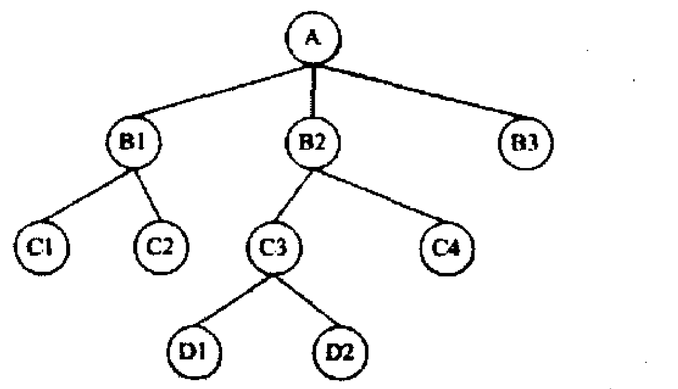


图 5-1　一个树的示例

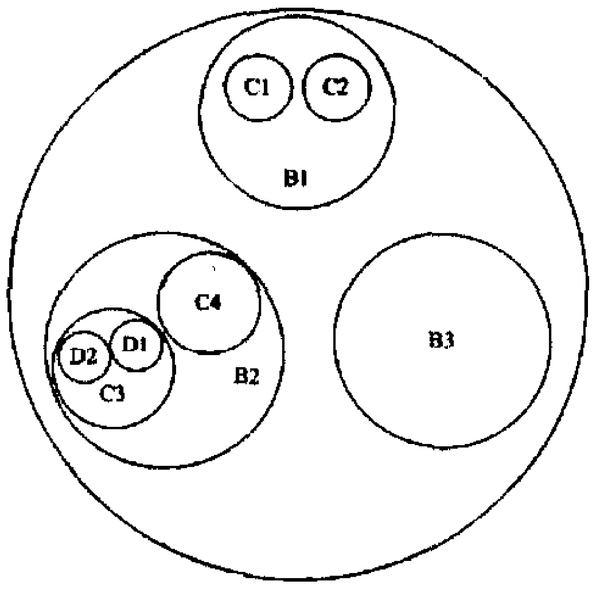
以递归定义分析图 5-1 所示的树，其由结点的有限集 $T=\{R,A,B,C,D,E,F\}$ 所构成，其中 R 为根结点，T 中其余结点可分成两个互不相交的子集：$T_1=\{A,C,D,E\}$，$T_2=\{B,F\}$；T_1 和 T_2 是根 R 的两棵子树，且本身又都是一棵树，例如 T_1，其根是 A，其余结点可分为三个互不相交的子集 $T_{11}=\{C\}$、$T_{12}=\{D\}$和 $T_{13}=\{E\}$，它们都是 A 的子树，其本身又都是只含一个根结点的树。对 T_2 亦可以进行类似的分析。

因此，采用子树的概念递归定义树为：树是由根结点和若干棵子树构成的。

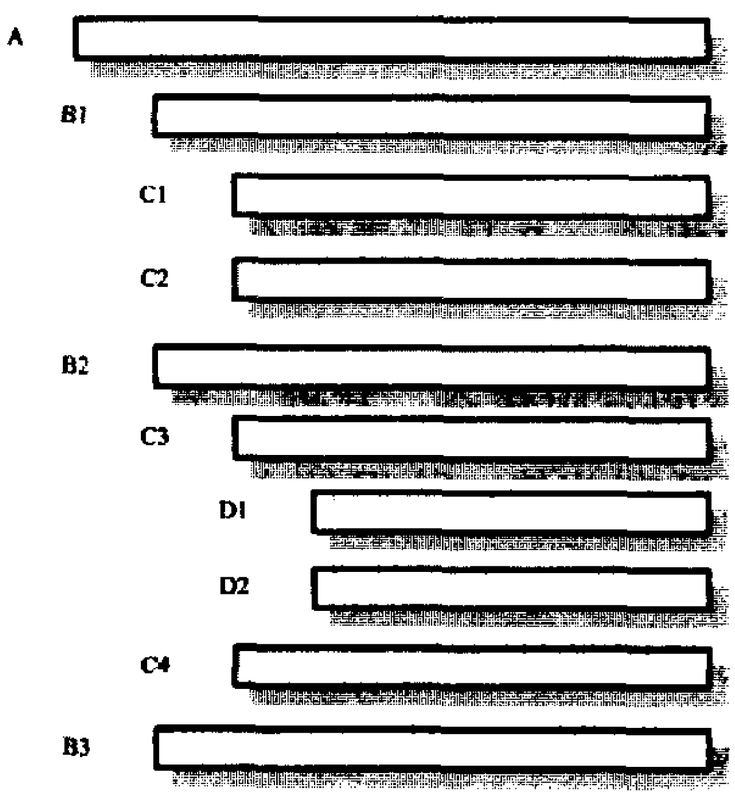
在不同的应用场合，树的表示方法可以不同。常见的是树型表示法、文氏图表示法、凹入图表示法以及广义表表示法等，如图 5-2 所示。



(a) 树型表示法



(b) 文氏图表示法



(c) 凹入图表示法

(A(B1(C1,C2),B2(C3(D1,D2),C4),B3))

(d) 广义表表示法

图 5-2　树的各种表示法

例如图 5-2(a)所示为树的树型表示法，此图所示的树可以用图 5-2(b)、(c)和(d)来表示，其中(b)是用集合的包含关系来描述树结构，即文氏图，(c)类似于常见的书籍目录，(d)则是用广义表的形式表示。

5.1.2　基本术语

一个结点的子树个数称为该结点的度(degree)。一棵树中结点度的最大值称为该树的度。度为零的结点称为叶子(leaf)或者终端结点。如图 5-1 中，结点 *A*、*B*、*C* 的度分别为 3、1、0。树的度为 3，*C*、*D*、*E*、*F* 均为叶子。度不为零的结点称为分支结点或者非终端结点，除根结点之外的分支结点统称为内部结点。

树中结点的后继结点称为儿子(child)或者儿子结点，简称儿子；结点的前趋结点称为儿子的双亲(parents)或者父亲结点，简称父亲。如图 5-2(a)中，*B*2 的后继结点为 *C*3、*C*4，所以 *C*3、*C*4 是 *B*2 的儿子，而 B2 则是 *C*3、*C*4 的父亲。同一个父亲的儿子互称为兄弟(sibling)，如图 5-2(a)中，*C*3、*C*4 互为兄弟。

若树中存在一个结点序列 $k_1k_2k_3\cdots k_j$，使得 k_i 是 k_{i+1} 的父亲($1\leqslant i<j$，则称该结点序列是从 k_1 到 k_j 的一条路径(path)或者道路。路径的长度等于 $j-1$，它是该路径所经过的边(即连接两个结点的线段)的数目。由此定义可知，若一个结点序列是路径，则在树型图表示中，该结点序列是“自上而下”地通过路径上的每条边。如图 5-2(a)中，结点 *A* 到 *D*1 有一条路径 *AB*2*C*3*D*1，其长度为 3。从图中可以看出，从树的根结点到树中其余结点均存在一条路径。但是结点 *C*1 和 *D*1 之间不存在路径，因为既不可能以 *C*1 为出发点“自上而下”地经过若干结点到达 *D*1，也不可能以 *D*1 为出发点“自上而下”地经过若干结点到达 *C*1。

若树中结点 *k* 到 k_s 存在一条路径，则称 *k* 是 k_s 的祖先(Ancestor)，k_s 是 k 的子孙(Descendant)。一个结点的祖先是从根结点到该结点路径上所经过的所有结点，而一个结点的子孙则是以该结点为根的子树中的所有的结点。

结点的层数(level)是从根开始算起的。设根结点的层数为 1，其余结点的层数等于其父亲结点的层数加 1。如图 5-2(a)中，*A* 的层数为 1，*B*1、*B*2、*B*3 的层数为 2，*C*1、*C*2、*C*3、*C*4 的层数为 3，*D*1、*D*2 的层数为 4。树中结点的最大层数称为树的高度(Height)或者深度(Depth)。则图 5-2(a)所示的树的高度为 4。

若把树中每个结点的各子树看成从左到右有次序的(即不能互换)，则称该树为有序树(Ordered Tree)，否则称为无序树(Unordered Tree)。

森林(Forest)是 $m(m\geqslant 0)$棵互不相交树的集合。如图 5-2(a)所示，如果删除了根结点 *A*，就得到三棵子树构成的森林；反之，把 *m* 棵独立的树看作是子树并加上一个根结点，则森林就变成了树。

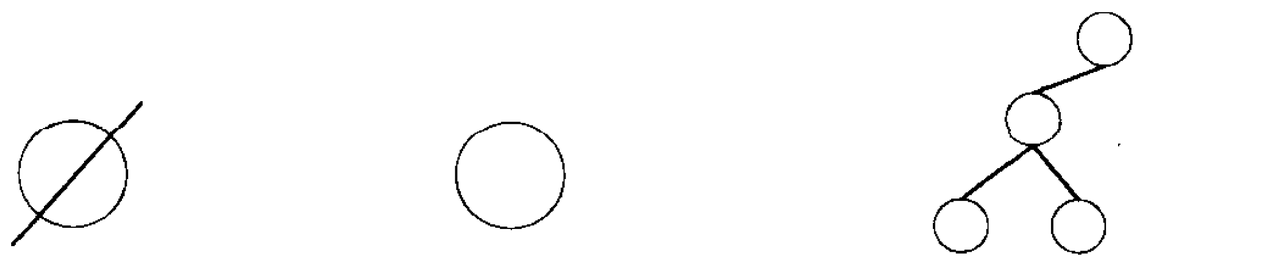
树中结点之间所存在的父子关系可以用于描述树型结构的逻辑特征：树中任一个结点都可以有零个或者多个后继(即儿子)结点，但至多只能有一个前趋(父亲)结点。树中只有根结点无前趋，叶子结点无后继。显然，这种父子关系是非线性的，所以树型结构是非线性

结构。祖先与子孙的关系则是对父子关系的延伸，其定义了树中结点的纵向次序。有序树的定义使得同一组兄弟结点之间是从左到右长幼有序的。对这一关系进行延伸，如果规定 k_1 和 k_2 是兄弟，且 k_1 在 k_2 的左边，则 k_1 的任一子孙都在 k_2 的任一子孙的左边，从而定义了树中结点的横向次序。

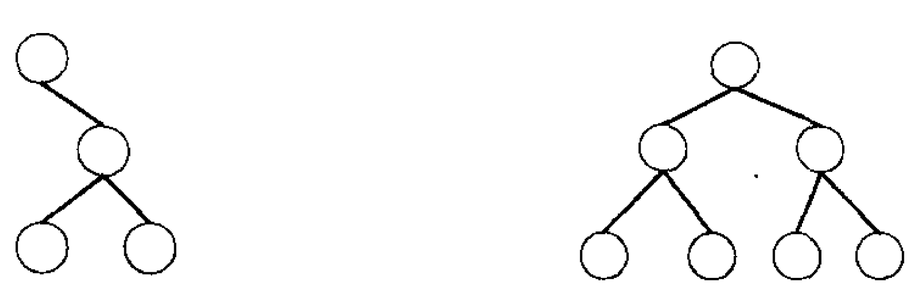
5.2　二叉树的定义

二叉树是由 $n(n \geq 0)$结点组成的有限集合，此集合或者为空，或者由一个根结点加上两棵分别称为左、右子树的，互不相交的二叉树组成。

从以上递归定义中可以看出，二叉树可以为空集，因此根可以有空的左子树或者右子树，亦或者左、右子树皆为空，如图 5-3 所示。



(a) 空二叉树　(b) 仅有一个根结点的二叉树　(c) 右子树为空的二叉树



(d) 左子树为空的二叉树　(e) 左、右子树均不为空的二叉树

图 5-3　二叉树的 5 种形态

二叉树举例，如图 5-4 所示。结点 A 是根结点，B、C 是 A 的子结点。结点 B 与 D 构成一棵子树。B 的两个子结点：左子树是空树，右子结点是 D。结点 A、C 和 E 是 G 的祖先，结点 D、E 和 F 的层数为 3，结点 A 的层数为 1。从 A 到 G 经过 C、E 两个顶点三条边，形成一条长度为 3 的路径。结点 D、G、H 和 I 是叶结点，A、B、C、E 和 F 是内部结点。结点 I 的深度为 4。这棵树的高度为 4。

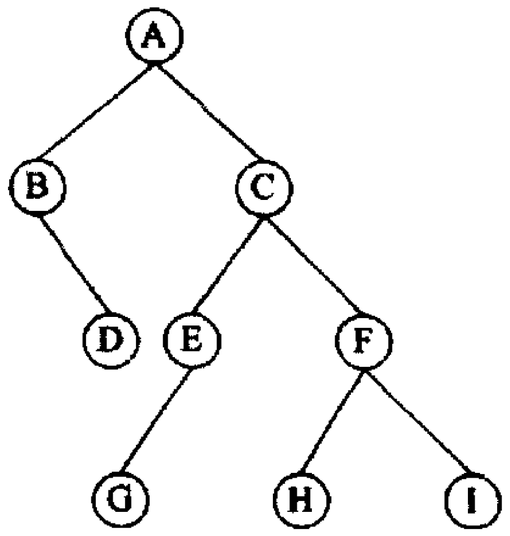
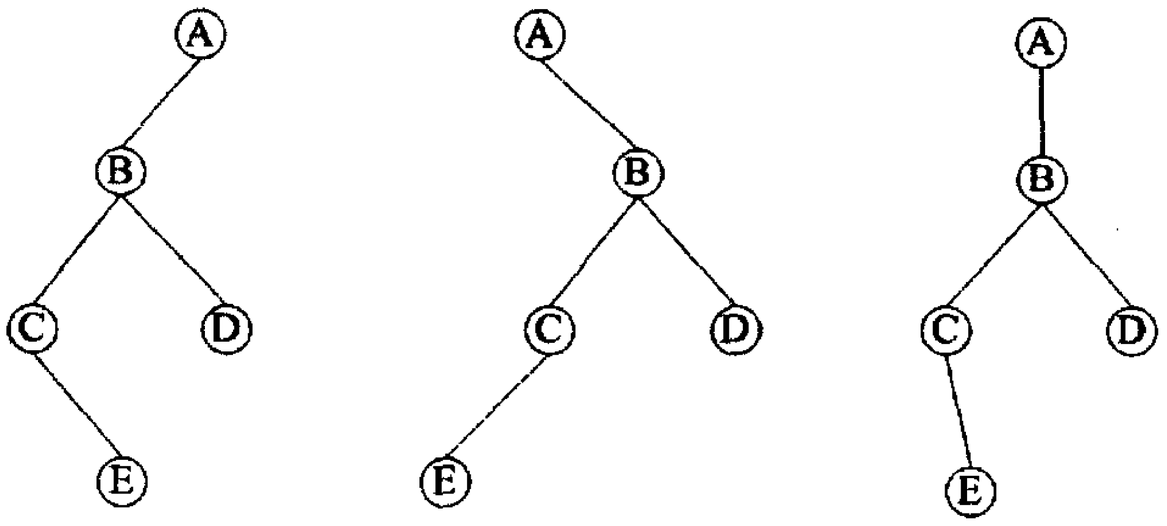


图 5-4　二叉树举例

从二叉树定义中可以看出，二叉树结构与一般树结构区别如下：

(1) 二叉树可以为空树，即不包含任何结点；一般树至少应有一个结点。

(2) 二叉树区别于度数为 2 的有序树，在二叉树中允许某些结点只有右子树而没有左子树；而有序树中，一个结点如果没有第一子树就不可能有第二子树的存在，如图 5-5 所示。



(a) 两棵不同的二叉树　　(b) 一般树

图 5-5　二叉树与一般树

因此，二叉树并非是树的特殊情形，它们是两种不同的数据结构。

5.3　二叉树的性质

本节讲述的主要内容是二叉树的基本性质以及满二叉树与完全二叉树的异同。

5.3.1　二叉树性质

性质 1　二叉树第 $i(i\geqslant 1)$层上的结点数最多为 2^{i-1}。

证明：第一层有一个结点，第二层最多有两个结点，第三层最多有四个结点，以此类推。数学归纳法证明如下：

归纳基础：$i=1$ 时，有 $2^{i-1}=2^0=1$。因为第一层上只有一个结点，即根结点，所以命题成立。

归纳假设：假设对所有的 $j(1\leqslant j<i)$命题成立，即第 j 层上至多有 2^{j-1} 个结点，需要证明 $j=i$ 时命题亦成立。

归纳步骤：根据归纳假设，第 $i-1$ 层上至多有 2^{i-2} 个结点。由于二叉树的每个结点至多有两个儿子，故第 i 层上的结点数至多是第 $i-1$ 层上的最大结点数的 2 倍，即 $j=i$ 时，该层上至多有 $2\times 2^{i-2}=2^{i-1}$ 个结点，因此命题成立。

性质 2　高度为 k 的二叉树最多有 2^k-1 个结点。

根据性质 1，将各层的结点个数相加，即可推导出性质 2。即

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

性质 3　对任何二叉树 T，设 n_0、n_1、n_2 分别表示度数为 0、1、2 的结点个数，则 $n_0=n_2+1$。

证明：因为二叉树中所有结点的度数均不大于 2，所以结点总数(记为 n)应该等于 0 度结点数(n_0)、1 度结点数(n_1)和 2 度结点数(n_2)之和：

$$n=n_0+n_1+n_2$$

再者，1 度结点有一个儿子，2 度结点有两个儿子，所以二叉树中儿子结点的总数是 n_1+2n_2，二叉树中只有根结点不是任何结点的儿子，因此二叉树中的结点总数可以表示为：

$$n=n_1+2n_2+1$$

由上述公式可得：

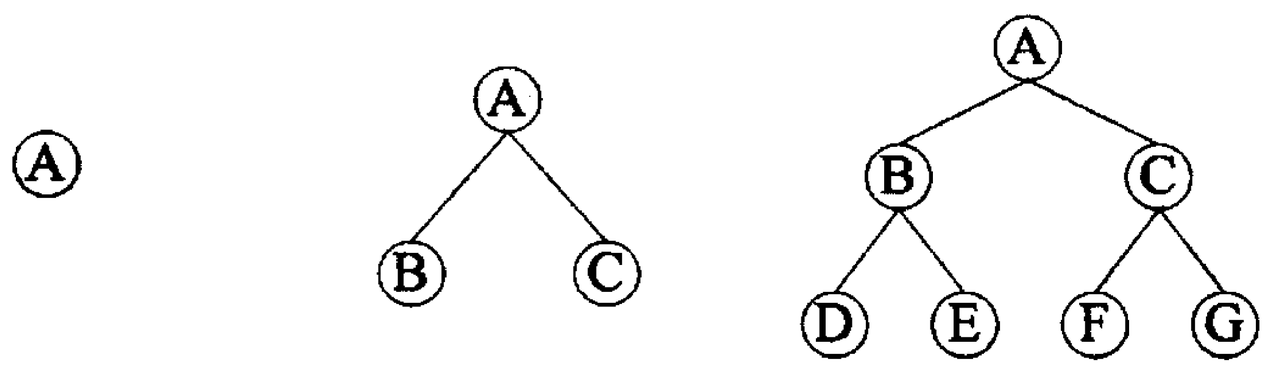
$$n_0=n_2+1$$

由性质 3 可以推导出：当 $n_2=0$ 时，$n_0=1$。即该二叉树有一个起始结点(根)和一个终端结点(叶子)，其他各结点有一个父亲一个儿子，二叉树退化为单链表。

满二叉树和完全二叉树是二叉树的两种特殊情形。

一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。

如图 5-6 所示是深度分别为 1、2、3 的满二叉树。满二叉树的特点是每一层上的结点数都达到最大值，即对给定的深度，它是具有最多结点数的二叉树。满二叉树不存在度数为 1 的结点，每个分支结点均有两棵高度相同的子树，且树叶都在最下一层上。

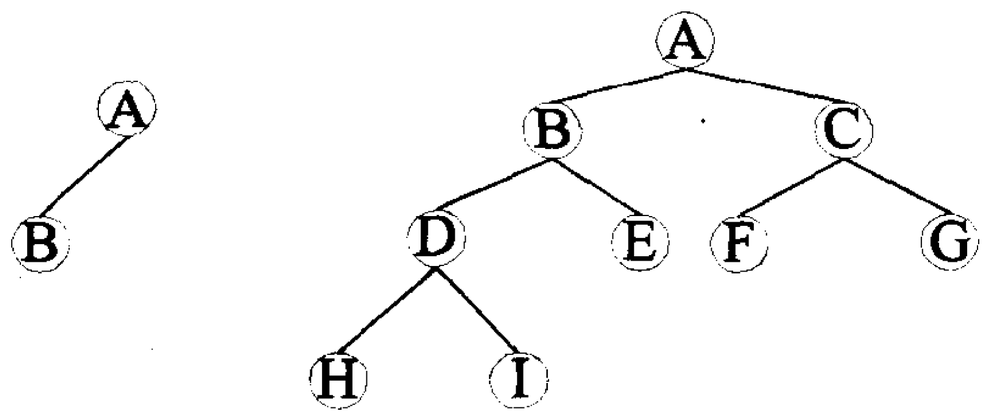


(a) 深度为 1 的满二叉树　(b) 深度为 2 的满二叉树　(c) 深度为 3 的满二叉树

图 5-6　三种不同深度的满二叉树

若一棵二叉树至多只有最下面的两层结点的度数可以小于 2，并且最下一层上的结点都集中在该层最左边的若干位置上，则此二叉树称为完全二叉树。

如图 5-7 所示为完全二叉树示例。



(a)　　(b)

图 5-7　完全二叉树

由定义及示例可以看出满二叉树是完全二叉树，但完全二叉树不一定是满二叉树。在满二叉树的最下一层上从最右边开始连续删去若干结点后得到的二叉树仍然是一棵二叉树。因此，在完全二叉树中，若某个结点没有左儿子，则它一定没有右儿子，即该结点必是叶结点。如图 5-8 所示，结点 F 没有左儿子而有右儿子 L，故它不是一棵完全二叉树。

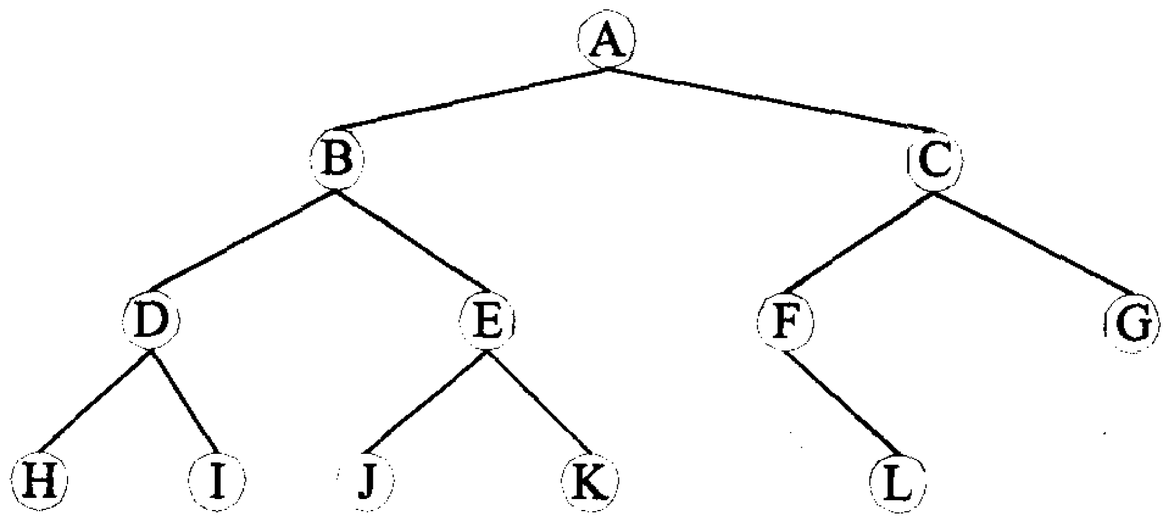


图 5-8　一棵非完全二叉树

性质 4　具有 n 个结点的完全二叉树(包括满二叉树)的高度为 $\lfloor \log_2 n \rfloor + 1$(或者 $\lceil \log_2 (n+1) \rceil$)。

证明：设所求完全二叉树的深度为 k，由完全二叉树的定义知道，其前 $k-1$ 层是深度为 $k-1$ 的满二叉树，一共有 $2^{k-1}-1$ 个结点。由于完全二叉树深度为 k，故第 k 层上还有若干个结点，因此，该完全二叉树的结点个数 $n > 2^{k-1}-1$。再者，由性质 2 知道 $n \leqslant 2^k - 1$，即

$$2^{k-1} - 1 < n \leqslant 2^k - 1$$

由此推导得 $2^{k-1} \leqslant n < 2^k$，取对数后有：

$$k - 1 \leqslant \log_2 n < k$$

因为 k 为整数，所以有 $k-1=\lfloor \log_2 n \rfloor$，即可得 $k=\lfloor \log_2 n \rfloor + 1$。

性质 5　满二叉树原理 非空满二叉树的叶结点数等于其分支结点数加 1。

证明：对 n(分支结点个数)做数学归纳法。

归纳基础：没有分支结点的非空二叉树有一个叶结点。有一个分支结点的满二叉树有两个叶结点，即当 $n=0$ 及 $n=1$ 时此定理成立。

归纳假设：设任意一棵有 $n-1$ 个分支结点的满二叉树有 n 个叶结点。

归纳步骤：假设树 T 有 n 个分支结点，取一个左右子结点均为叶结点的分支结点 I。去掉 I 的两个子结点，则 I 成为叶结点，把新树记为 T'，T'有 $n-1$ 个分支结点，根据归纳假设，T' 有 n 个叶结点，现在把两个叶结点归还给 I。从而得到 T 有 n 个分支结点。既然 T' 有 n 个叶结点，再加上两个则得到 T 有 $n+2$ 个叶结点，但是在 T'中结点 I 被计算为叶结点，而现在则是分支结点，于是树 T 有 $n+1$ 个叶结点和 n 个分支结点。

因此，根据归纳原理，定理对于任意 $n \geqslant 0$ 成立。

性质 6　一棵非空二叉树空子树的数目等于其结点数目加 1。

证明 1　设二叉树 T，将其所有空子树换成叶结点，把新的二叉树记为 T'。所有原来的树 T 的结点现在是树 T' 的分支结点。由于树 T 的所有分支结点都有两个子结点，并且树 T 中的每个叶结点在树 T' 中都有两个叶结点，所以树 T'是满二叉树。根据满二叉树定理，新添加的叶结点数目等于树 T 的结点数目加 1，而每个新添加的叶结点对应树 T 的一棵空子树，因此树 T 中空子树的数目等于树 T 中结点数目加 1。

证明 2　根据定义，树 T 中每个结点都有两个子结点，因此一棵(实际上)有 n 个结点的二叉树有 $2n$ 个子结点。除了根结点以外，每个结点都有一个父结点，于是共有 $n-1$ 个父结点，即有 $n-1$ 个非空子结点。既然子结点数目为 $2n$，则其中有 $n+1$ 个为空。

5.3.2　二叉树的抽象数据类型

下列给出一个二叉树结点的 Java 接口，称之为 BinNode。BinNode 类中存储了指向 Object 类的引用。创建二叉树时，可以根据需要而采用实际的数据类型。成员函数包括返回元素的值，返回左、右结点指针，设置元素的值，判断该结点是否为叶结点。

```
interface BinNode { //二叉树结点的抽象数据类型
//返回并设置元素值
public Object element();
public Object setElement(Object v);

//返回并设置左孩子
public Binnode left();
public Binnode setLeft(BinNode p);

//返回并设置右孩子
public Binnode right();
public Binnode setRight(BinNode p);

//判断是否为叶结点
public boolean isLeaf();
}//interface BinNode
```

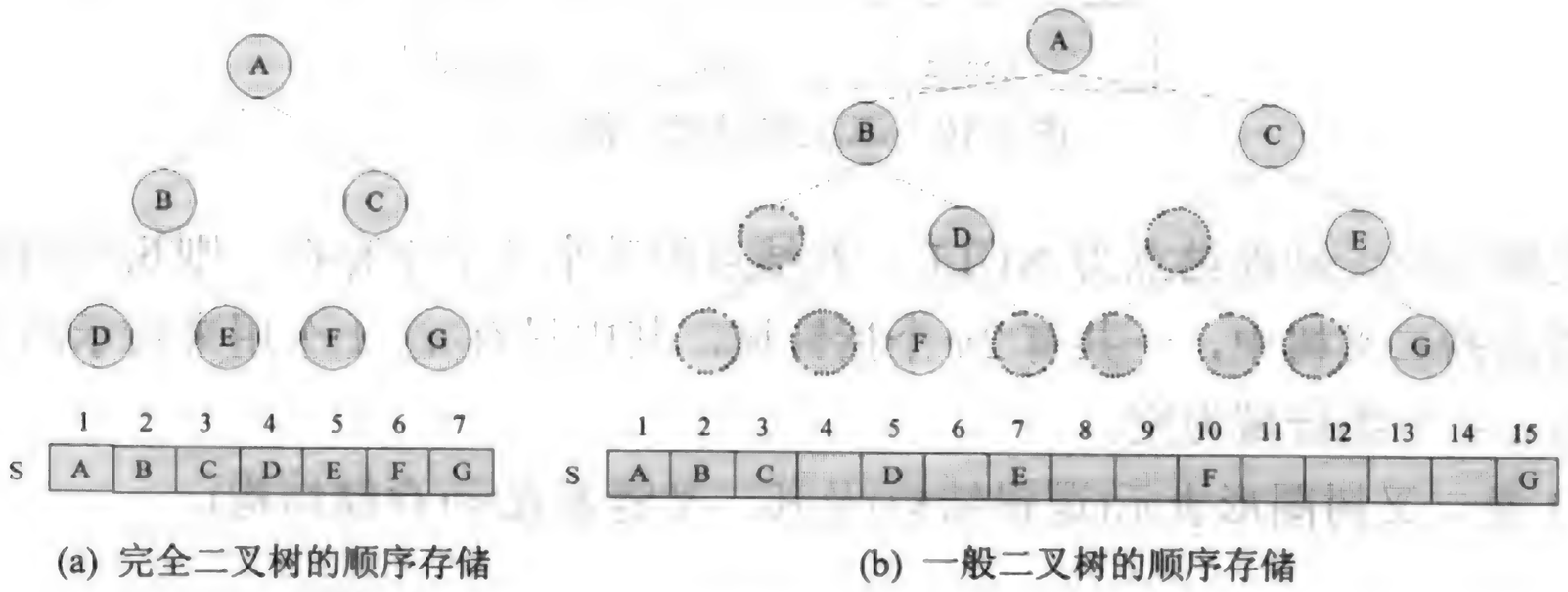
5.4　二叉树的存储结构

本节讲述的主要内容为二叉树的存储结构：顺序存储结构和链接存储结构，以及二叉树的实现。

5.4.1　二叉树的顺序存储结构

二叉树的顺序存储结构是把二叉树的所有结点按照一定的次序顺序存储到一组包含 n 个存储单元的空间中。在二叉树的顺序存储结构中只存储结点的值(数据域)，不存储结点之间的逻辑关系，结点之间的逻辑关系由数组中下标的顺序来体现。

二叉树顺序存储的原则是：不管给定的二叉树是不是完全二叉树，都看作完全二叉树，即按完全二叉树的层次次序(从上到下，从左到右)把各结点依次存入数组中。如图 5-9 所示为二叉树的顺序存储结构。



(a) 完全二叉树的顺序存储　　(b) 一般二叉树的顺序存储

图 5-9　二叉树顺序存储结构示意图

在顺序存储结构中，由某结点的存储单元地址可以推出其父亲、左儿子、右儿子及兄弟的地址，假设给定结点的地址为 I，则：

(1) 若 $I=1$，则该结点是为根结点，无父亲。

(2) 若 $I\neq1$，则该结点的父亲结点地址为 $I/2$ 的整数部分。

(3) 若 $2\times I\leqslant n$，则该结点的左儿子结点地址为 2×I，否则该结点无左儿子。

(4) 若 $2\times I+1\leqslant n$，则该结点的右儿子结点地址为 $2\times I+1$，否则该结点无右儿子。

(5) 若 I 为奇数(不为 1)，则该结点的左兄弟为 $I-1$。

(6) 若 I 为偶数(不为 n)，则该结点的右兄弟为 $I+1$。

在图 5-9(a)中，结点 C 的地址 $I=3$，其父亲 A 在 S[i/2](即 S[1])中，其左儿子 F 在 S[2*I](即 S[6])中，其右儿子 G 在 S[2*I+1](即 S[7])中，其右兄弟 B 在 $S[I-1]$(即 S[2])中。

而在图 5-9(b)中，结点 C 的地址 $I=3$，其左儿子应在 S[6]中，而 S[I]的内容为空，故结点 C 没有左儿子；结点 F 的地址 $I=10$，其左、右儿子应分别在 S[20]和 S[21]中，但 20 已经超出数组的下标范围，故结点 F 没有儿子。

显然地，顺序存储结构对完全二叉树而言，既简单又节省存储空间。对于一般二叉树，为了能用结点在数组中的相对位置表示结点之间的逻辑关系，也必须按完全二叉树的形式来存储树中的结点，这必然造成存储空间的浪费，这是因为顺序结构中存储很多空结点，而且随二叉树高度的增大，空结点的数量也会急速增多。如果采用存储压缩的方法把表中的空结点压缩掉，又必然会给二叉树的访问、插入、删除等带来极大的不便。在最坏的情况下，一个高度为 k 且只有 k 个结点的右单支树却需要 $2^k - 1$ 个结点的存储空间。

5.4.2　二叉树的链接存储结构

由于采用顺序存储结构存储一般二叉树造成大量存储空间的浪费，因此，一般二叉树的存储结构更多地采用链接的方式。

二叉树的链接存储中每个结点由数据域和指针域两部分组成。二叉树每个结点的指针域有两个，一个指向左儿子，一个指向右儿子，如图 5-10 所示。此时还需一个链表的头指针指向根结点。二叉树的链接存储结构也称为二叉链表。

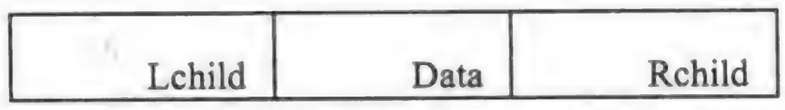


图 5-10　二叉树链接存储结构

若二叉树为空，则根结点为 NULL。若结点的某个儿子不存在，则相应的指针为空。具有 n 个结点的二叉树中，一共有 $2n$ 个指针域，其中只有 $n-1$ 个用来指示结点的左右儿子，其余的 n+1 个指针域为空。

图 5-11 是二叉树图形表示(逻辑结构)及其二叉链表表示(存储结构)。

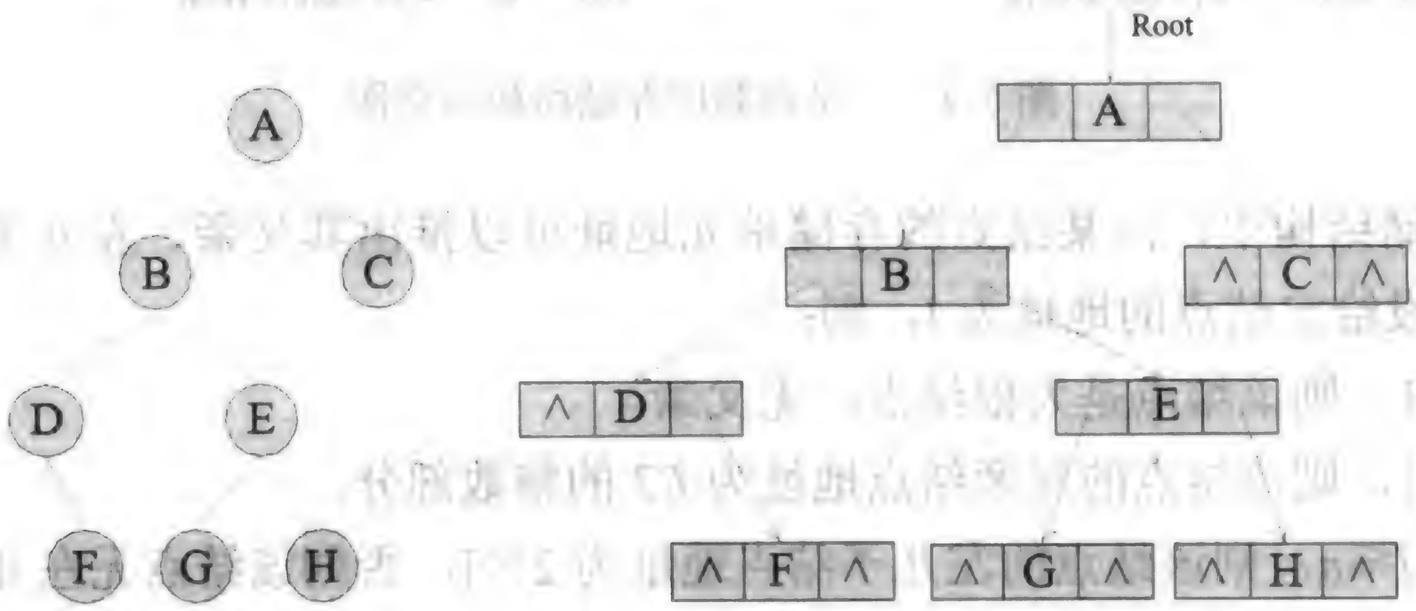


图 5-11　二叉树的二叉链表表示

下面给出一个二叉树结点类 BinNodePtr 的声明，它表示二叉树结点采用包含一个数据区和两个指向子结点的指针结构时的成员函数的说明。

```
//具有指向左右子结点指针的二叉树
class BinNodePtr implements BinNode {
  private object element; //结点对象
  private BinNode left; //左儿子指针
```

```
    private BinNode right; //右儿子指针

public BinNodePtr() {left = right =null; } //创建结点 1
public BinNodePtr(Object val) { //创建结点 2
    left = right =null;
element = val;
}

public BinNodePtr(Object val, BinNode l, BinNode r)   //创建结点 3
{ left = l; right = r; element = val; }

//返回和设置元素值
public Object element() { return element; }
public Object setElement(Object v) { return element = v; }

//返回和设置左儿子
public BinNode left() { return left; }
public BinNode setLeft(BinNode p) { return left = p; }

//返回和设置右儿子
public BinNode right() { return right; }
public BinNode setRight(BinNode p) { return right = p; }

//判断叶子结点
public boolean isLeaf()
{ return (left = null) && (right = null); }
}//class BinNodePtr
```

5.4.3　二叉树的实现举例

二叉树的实现原则为：

- 以第一个建立的元素为根结点。
- 依次序将元素值与根结点做比较，若元素值大于根结点值，则将元素值往根结点的右子结点移动，若此右子结点为空，则将元素值插入；否则就重复比较，直到找到适当的空结点为止。若元素值小于根结点值，则将元素值往根结点的左子结点移动，若此左子结点为空，则将元素值插入；否则就重复比较，直到找到适当的空结点为止。

1. 以数组方式实现二叉树

先依次序输入元素值，一一建立二叉树数组，其中根结点的下标为 1，其余结点的建

立则遵循左小(level*2)右大(level*2+1)的原则，最后输出所建立二叉树的结点内容。

```
import ConsoleReader.*; //引入数据输入类

public class bitree01
{
 public static void main (String args[])
 {
      int i; //循环变量
      int Index=1; //数组下标变量
      int Data; //读取输入值的临时变量
      BiTreeArray BiTree=new BitreeArray(); //声明二叉树数组
      System.out.printin("请输入二叉树数据元素(输入 0 退出！ ): ");

      ConsoleReader console=new ConsoleReader(System.in);
      do //依次序读取结点值
      {
           System.out.print("Data"+Index+" : ");
           Data=console.readInt();
           Bitree.Create(Data); //建立二叉树
           Index++;
      }while(Data!=0);
           BiTree.PrintAll(); //输出二叉树的结点值
 }
}

class BiTreeArray
{
 int MaxSize=16;
 int[] ABiTree=new int[MaxSize];

 public void BiTreeArray()
 {
      int i;
      for (i=0;i<MaxSize;i++)
      ABiTree[i]=0;
 }

//建立二叉树
 public void Create(int Data)
 {
      int i;
      int Level; //树的层数

      Level=1; //从层 1 开始建立
```

```
while(ABiTree[Level]!=0) //判断是否存在子树
        {
              if Data<ABiTree[Level]) //判断是左子树还是右子树
                 Level=Level*2; //左子树
              else
                 Level=Level*2+1; //右子树
        }
              ABiTree[Level]=Data; //将元素值插入结点

//输出二叉树结点值
 public void PrintAll()
 {
        int i;
        System.out.println("二叉树结点值依次是：");
        for (i=1;i<MaxSize;i++)
        {
              System.out.print("Node"+i);
              System.out.println(":[ "+ABiTree[i]+ "]");
        }
 }
```

2. 以数组方式实现二叉树的链接存储

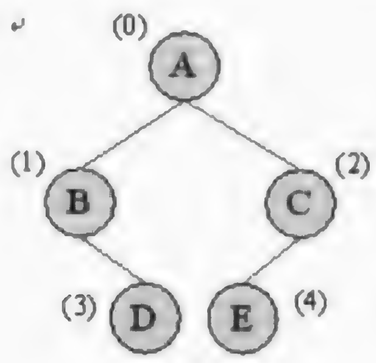
定义一个类，该类包含三个字段，一个字段用于存放结点的数据值，另两个字段分别用于存放左儿子结点和右儿子结点在数组中的下标。

结点数组的元素结构为：

lchild	data	rchild

其中，data 为存放结点的数据值；lchild 为存放左儿子结点在数组中的下标；rchild 为存放右儿子结点在数组中的下标。

在结点数组中，会将根结点置于数组结构中下标为 0 处，将结点值存在 data 字段，而 lchild 及 rchild 字段则分别存储左右子结点在数组结构中的下标，若子结点不存在则存值 - 1。例如，有一棵二叉树的树状结构与结点数组表示法如图 5-12 所示。



下标	lchild	data	rchild
0	1	A	2
1	-1	B	3
2	4	C	-1
3	-1	D	-1
4	-1	E	-1

图 5-12　二叉树的树状结构及其数组表示法

其中根结点为 *A*，*A* 在结点数组下标为 0 处，其左子结点 *B* 在下标为 1 处，右子结点在下标为 2 处，故结点 *A* 的 1child 字段和 rchild 字段分别是 1 和 2。而结点 C 的 rchild 字段值为 - 1，表示其没有右子结点。而结点 *D* 和 *E* 都是叶结点(1eaf node)没有子结点，故 1child 和 rchild 字段均为 - 1。

以下示例为以结点数组方式建立二叉树，并输出结点内容。依次序输入结点值，并存入数组中，再一一建立成二叉树数组，其中根结点的下标为 0，其余结点的建立则遵守左字段存左子结点的下标，右字段存右子结点下标的原则，最后输出所建立二叉树的结点值。

```
import ConsoleReader.*; //引入已定义的数据输入类

public class bitree02
{
    public static void main(String argS[])
    {
        int i; //循环变量
        int index=l; //数组下标变量
        int data; //输入值所使用的临时变量
        BiTreeArray BiTree=new BiTreeArray(); //声明二叉树数组

        System.out.println("请输入二叉树结点值(输入 0 退出 0)：");
        ConsolReader    console=new ConsoleReader(SyStem.in);

        System.out.print("Data"+index+" ：    ");
        Data=console.readInt();
        BiTree.TreeData[0]=data;
        index++;

        while (true) //读取输入值
        {
            System.out.print("Data "+index+" ：    ");
            data=console.readInt();
            if (data==0)
               break;
            BiTree.Create(data); //建立二叉树
            index++;
        }
        BiTree.PrintAll(); //输出二叉树的内容
    }
}

class BiTreeArray
{
    int MaxSize=16;
    int[]    TreeData=new int[MaxSize];
    int[]    RightNode=new int[MaxSize];
```

```
int[]   LeftNode=new int[MaxSize];

 public BiTreeArray()
 {
    int i; //循环变量
    for (i=0; i<MaxSize; i++)
    {
       TreeData[i]=0;
       RightNode[i]=-1;
       LeftNode[i]=-1;
    }
 }

//建立二叉树
    public void Create(int data)
    {
        int i;
        int level=0; //树的层数
        int Position=0;

        for (i=0; TreeData[i]!=0; i++)
        TreeData[i]=data;
        while (true) //寻找结点位置
        {
            //判断是左子树还是右子树
            if (data > TreeData[Level])
                //右子树是否有下一层
                if (RightNode[level]!=-1)
                   level=RightNode[level];
                else
                {
                    Position=-1; //设置为右子树
                    Break;
                }
                else
                {
                  //左子树是否有下一层
                  if (LeftNode[level]!=-1)
                     level=LeftNode[level];
                  else
                  {
                      Position=1; //设置为左子树
                      break;
                  }
```

```
                }
            }
            if (Position==1) //建立结点的左右连接
               LeftNode[level]=i; //连接左子树
            else
               RightNode[level]=i; //连接右子树
        }

    //打印所有二叉树结点值
        public void PrintAll()
        {
           int i;

           System.out.println("二叉树结点值： ");
           System.out.println("      [lchild]   [data]   [rchild] ");
           for (i=0;i<MaxSize;i++)
           {
              if(TreeData[i]!=0)
              {
                  System.out.print("Node"+i);
                  System.out.print("      ["+LeftNode[i]+ "]");
                  System.out.print("    ["+TreeData[i]+ "]");
                  System.out.println("    ["+RightNode[i]+ "]");
              }
           }
        }
```

以上程序的结构可以改善二叉树数组表示法中，若要插入或删除结点需要移动大量数据的问题。因为有两个字段 lchild 和 rchild 来存储左右子树的下标，所以插入或删除结点时只要改变这两个字段值，而不需要大量移动数据。

5.5　二叉树的遍历

本节讲述的主要内容为二叉树的前序遍历、中序遍历以及后序遍历的方法。

“遍历”是抽取数据结构中的各个数据值，例如：数组和链表可从前端到尾端或从尾端至前端依序抽取各个数据值。而二叉树是一种特殊的数据结构，每个结点下又各有左、右两个分支。“二叉树的遍历”是以固定的顺序，系统地抽取二叉树中的各结点，且每个结点均恰好被抽取一次。

二叉树中每个结点均有左右两个分支，在遍历的过程中可以选择往左或往右走，遍历

结束，每个结点恰被抽取一次。事实上，二叉树的遍历是以递归的方式进行，依递归的调用顺序的不同，可分为 3 种不同的遍历方式：

- 前序遍历方式
- 中序遍历方式
- 后序遍历方式

5.5.1　二叉树的前序遍历

前序遍历(Preorder Traversal)是先遍历根结点，再遍历左子树，最后才遍历右子树。即若二叉树非空，则依次进行如下操作：

(1) 访问根结点；

(2) 前序遍历左子树；

(3) 前序遍历右子树。

例如图 5-13 所示。

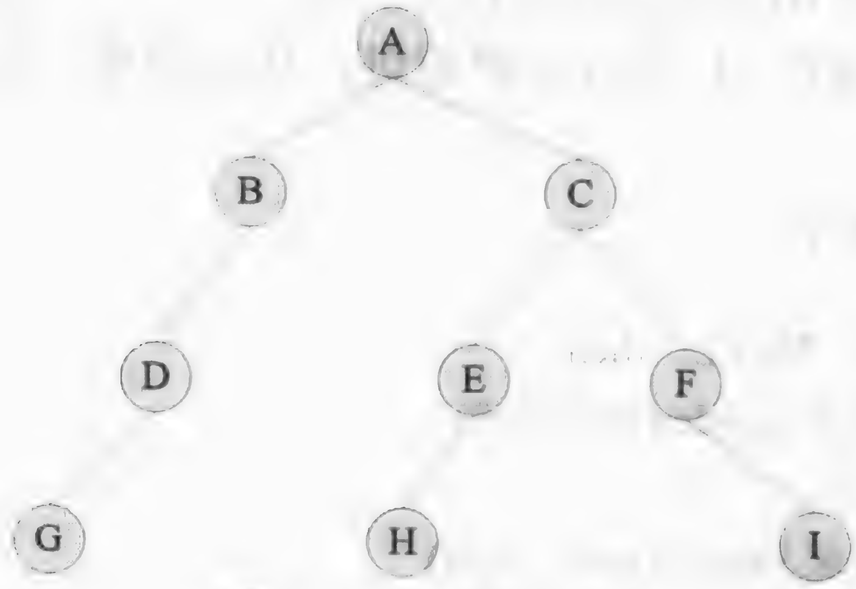


图 5-13　一棵二叉树

图 5-13 中处理完根结点 *A* 后，先往左子树经过 *B* 再到 *D*，由于 *D* 有左子树，故继续转向左子树 *G*。再回到 *B*，因为 *B* 没有右子树，所以此时 *A* 的左子树均遍历完毕，则转向 *A* 的右子树，先到 *C*，再往左边继续遍历，依此类推，故可得到前序遍历的顺序为 ABDGCEHFI。

前序遍历的递归算法如下：

```
if  指向根结点的指针=NULL  then
    (1) 处理当前的结点
    (2) 往左走,递归处理 preorder(root->left)
    (3) 往右走,递归处理 preorder(root->right)
    此为空树,遍历结束
else
```

Java 语言实现示例为：

```
void preorder(BinNode rt) //rt 是子树的根
{
```

```
    if (rt == null) return; //空子树
    visit(rt);
    preorder(rt.left());
    preorder(rt.right());
}
```

5.5.2　二叉树的中序遍历

中序遍历(Inorder Traversal)是先遍历左子树，再遍历根结点，最后才遍历右子树。即若二叉树非空，则依次进行如下操作：

(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。

以图 5-13 为例，从结点 *A* 开始，一直往左走到 *G* 无法再前进，则处理 *D*。此时已遍历完 *B* 的左子树，接着处理 *B*，再往 *B* 的右方前进。由于 *B* 没有右子树，故 *A* 的左子树遍历完毕，可处理节点 *A*，再往 *A* 的右子树前进，依此类推，故可得到中序遍历的次序为 GDBAHECFI。

中序遍历的递归算法如下：

```
if   指向根结点的指针=NULL   then
    (1) 往左走,递归处理 inorder(root->left)
    (2) 处理当前的结点
    (3) 往右走,递归处理 inorder(root->right)
    此为空树,遍历结束
else
```

Java 语言实现示例为：

```
void inorder(BinNode rt) //rt 是子树的根
{
   if (rt == null) return; //空子树
   inorder(rt.left());
   visit(rt);
   inorder(rt.right());
}
```

5.5.3　二叉树的后序遍历

后序遍历(Postorder Traversal)是先遍历左子树，再遍历右子树，最后才遍历根结点。即若二叉树非空，则依次进行如下操作：

(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。

以图 5-13 为例，从结点 *A* 开始一直往左走到 *G* 无法再前进，由于 *G* 没有左、右子树，故处理结点 *G*。之后由于 *D* 的左子树遍历完毕，且无右子树，故进而处理 *D*，而 *B* 的左子树也相应地完成。且结点 *B* 没有右子树，故可接着处理 *B*。此时结点 *A* 的左子树已遍历完毕，可进而往 *A* 的右子树经过 *C* 往前进，依此类推，当 *A* 的右子树遍历完毕后，方可处理根结点 *A*，故可得到后序遍历的顺序为 GDBHEIFCA。

后序遍历的递归算法如下：

```
if   指向根结点的指针=NULL   then
    (1) 往左走,递归处理 postorder(root->left)
    (2) 往右走,递归处理 postorder(root->right)
    (3) 处理目前的结点
    此为空树,遍历结束
else
```

Java 语言实现示例为：

```
void postorder(BinNode rt) //rt 是子树的根
{
    if (rt == null) return; //空子树
    postorder(rt.left());
    postorder(rt.right());
    visit(rt);
}
```

5.5.4　二叉树的层次遍历

层次遍历是指从二叉树的第一层(根结点)开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

以图 5-13 为例，按层次遍历方式进行遍历所得到的的结果序列为：

A B C D E F G H I

由层次遍历的定义可以推知，在进行层次遍历时，对一层结点访问完后，再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问，就完成了对下一层从左到右的访问。因此，在进行层次遍历时，需设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队，然后从队头取出一个元素，每取出一个元素，执行两个操作：访问该元素所指结点；若该元素所指结点的左、右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。此过程循环进行，直至队列为空，表示二叉树的层次遍历结束。

所以，对一棵非空的二叉树进行层次遍历可按照如下步骤进行：

(1) 初始化一个队列；

(2) 二叉树的根结点放入队列；

(3) 重复步骤(4)~(7)直至队列为空；

(4) 从队列中取出一个结点 x；

(5) 访问结点 x；

(6) 如果 x 存在左子结点，将左子结点放入队列；

(7) 如果 x 存在右子结点，将右子结点放入队列。

5.6　线索二叉树

本节讲述的主要内容为二叉树线索化的概念以及前序线索二叉树、中序线索二叉树和后序线索二叉树。

5.6.1　二叉树的线索化

在线性结构中，各结点的逻辑关系是顺序的，寻找某一结点的前趋结点和后继结点很方便。对于二叉树，由于它是非线性结构，所以树中的结点不存在前趋和后继的概念，但当我们对二叉树以某种方式遍历后，就可以得到二叉树中所有结点的一个线性序列，在这种意义下，二叉树中的结点就有了前趋结点和后继结点。

二叉树通常采用二叉链表作为存储结构，在这种存储结构下，由于每个结点有两个分别指向其左儿子和右儿子的指针，所以寻找其左、右儿子结点很方便，但要找该结点的前趋结点和后继结点则比较困难。例如，要在中序遍历的前提下，寻找任一结点的前趋结点，如果该结点存在左儿子结点，那么从该左儿子结点开始，沿着右指针链不断向下找，当某结点的右指针域为空时，该结点就是所要寻找的前趋结点；如果某结点不存在左儿子结点，则需遍历二叉树才能确定该结点的前趋结点。

实际上，在大多数情况下，寻找结点的前趋结点或后继结点都需要遍历二叉树。

为方便寻找二叉树中结点的前趋结点或后继结点，可以通过一次遍历记下各结点在遍历所得的线性序列中的相对位置。保存这种信息的一种简单的方法是在每个结点增加两个指针域，使它们分别指向依某种次序遍历时所得到的该结点的前趋结点和后继结点，显然这样做要浪费相当数量的存储单元。如果仔细分析一棵具有 n 个结点的二叉树，就会发现，当它采用二叉链表作存储结构时，二叉树中的所有结点共有 $n+1$ 个空指针域。因此，可以设法利用这些空指针域来存放结点的前趋结点和后继结点的指针信息，这种附加的指针称为“线索”。我们可以作这样的规定，当某结点的左指针域为空时，令其指向依某种方式遍历时所得到的该结点的前趋结点，否则指向它的左儿子；当某结点的右指针域为空时，

令其指向依某种方式遍历时所得到的该结点的后继结点，否则指向它的右儿子。增加了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树(Threaded Binary Tree)。

这样作会产生如下的问题：一个结点的左、右指针量指向的是它的左、右儿子结点，还是它的前趋结点和后继结点?为了区分一个结点的指针是指向其儿子的指针还是指向其前趋或者后继的线索，可以在每个结点上增加两个线索标志域 ltag 和 rtag，这样线索链表的结点结构为：

lchild	ltag	data	rtag	rchild

其中：

左线索标志　ltag=0；//lchild 是指向结点的左儿子的指针

　　ltag=1；//lchild 是指向结点的前趋结点的左线索

右线索标志　rtag=0；//rchild 是指向结点的右儿子的指针

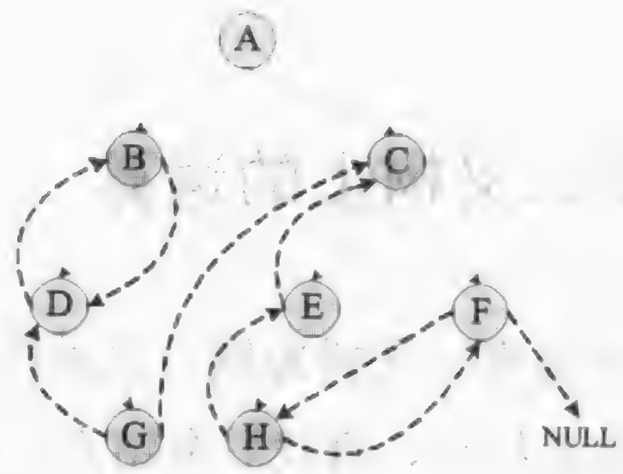
　　rtag=1；//rchild 是指向结点的后继结点的右线索

每个标志位令其只占一个 bit，这样就只需增加很少的存储空间。在结构示意图中，线索通常用虚线表示。

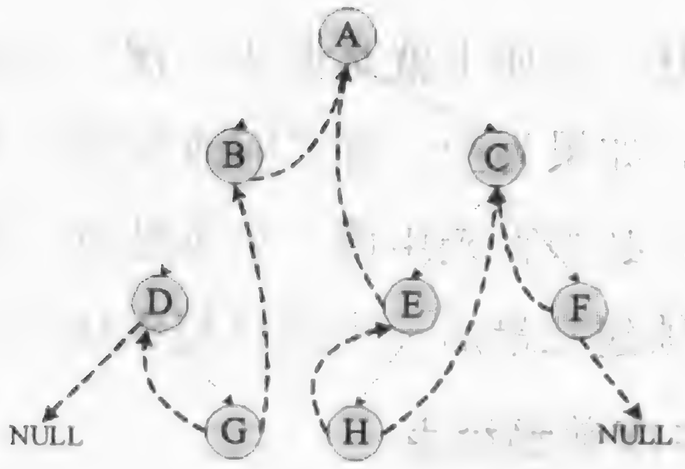
一棵二叉树以某种方式遍历并使其变成线索二叉树的过程称为二叉树的线索化。

对同一棵二叉树遍历的方式不同，所得到的线索树也不同，二叉树有前序、中序和后序 3 种遍历方式，所以线索树也有前序线索二叉树、中序线索二叉树和后序线索二叉树 3 种。图 5-14(a)所示的一棵二叉树，用三种方式线索化后得到的前序线索二叉树、中序线索二叉树和后序线索二叉树分别如图 5-14(b)、(c)、(d)所示。

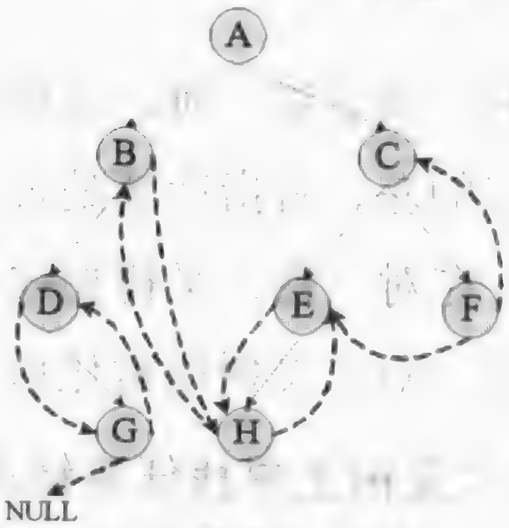
(a) 一棵二叉树



(b) 前序线索二叉树



(c) 中序线索二叉树



(d) 后序线索二叉树

图 5-14　线索二叉树

为讨论算法方便起见，通常在二叉树中增加一个与树中结点相同类型的头结点，令头结点的信息域为空，其 lchild 域指向二叉树的根结点，当二叉树为空时，lchild 域值为空；其 rchild 域指向以某种方式遍历二叉树时最后访问的结点，当二叉树为空时，rchild 域指向该结点本身，同时令原来指向二叉树根结点的头指针指向该头结点，以某种方式遍历二叉树时第一个被访问结点的左指针域和最后一个被访问结点的右指针域的值如果是线索，也指向该头结点。

如图 5-15 所示为一棵添加了头结点的中序线索二叉树。

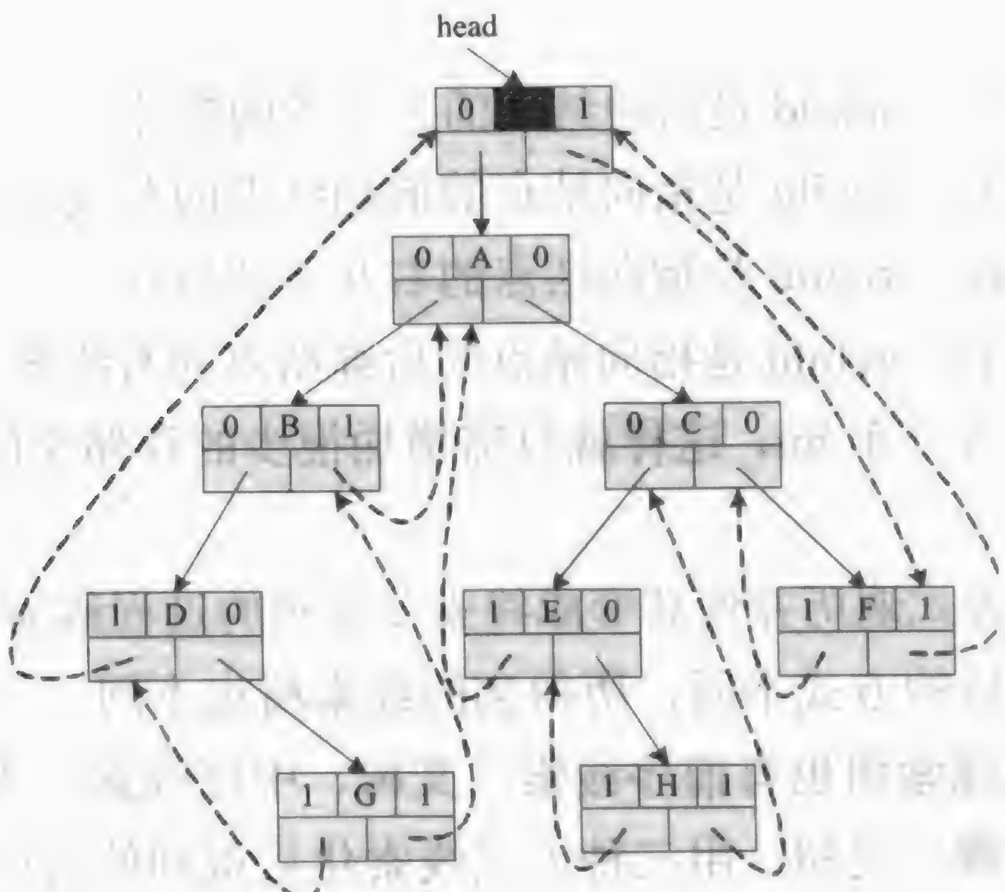


图 5-15　带头结点的中序线索二叉树

5.6.2　线索二叉树上的运算

下面以中序线索二叉树为例，讨论线索二叉树的建立、线索二叉树的遍历以及在线索二叉树上查找前趋结点、查找后继结点、插入结点和删除结点等操作的实现算法。

1. 建立一棵中序线索二叉树

建立线索二叉树，或者说，对二叉树线索化，实质上就是遍历一棵二叉树，在遍历的过程中，检查当前结点的左、右指针域是否为空，如果为空，将它们改为指向前趋结点或后继结点的线索。另外，在对一棵二叉树加线索时，必须首先申请一个头结点，建立头结点与二叉树的根结点的线索，对二叉树线索化后，还须建立最后一个结点与头结点之间的线索。

2. 在中序线索二叉树上寻找任意结点的中序前趋结点

对于中序线索二叉树上的任一结点，如果该结点的左标志为 1，那么其左指针域所指向的结点便是它的前趋结点；反之，如果该结点的左标志为 0，表明该结点有左儿子，根

据中序遍历的定义，它的前趋结点是以该结点的左儿子为根结点的子树的最右结点，即沿着其左子树的右指针链向下查，当某结点的右标志为1时，它就是所要找的前趋结点。

3. 在中序线索二叉树上寻找任意结点的中序后继结点

对于中序线索二叉树上的任一结点，如果它的右标志为 1，那么其右指针域所指向的结点就是它的后继结点；反之，如果该结点的右标志为 0，表明该结点有右儿子，在这种情况下，由中序遍历的定义可知，它的后继结点是以其右儿子为根结点的子树的最左结点，即沿着该结点右子树的左指针链向下找，当某结点的左标志为 1 时，该结点就是所要找的后继结点。

以上给出的仅是在中序线索二叉树中寻找某结点的前趋结点和后继结点的算法。在前序线索二叉树中找结点的后继结点以及在后序线索二叉树中找结点的前趋结点可以采用同样的方法分析和实现。在前序线索二叉树中找结点的前趋结点和在后序线索二叉树中找结点的后继结点的算法比较复杂，要分多种情况进行讨论。

4. 在中序线索二叉树中查找值为 x 的结点

利用在中序线索二叉树上寻找后继结点的算法，可以很容易地查找线索二叉树中的任一结点。设置一指针变量 p，开始时令 p 指向线索二叉树的根结点，若 p 所指结点的信息域值为 x，则查找成功，否则令 p 指向它原来所指结点的后继结点，继续检查 p 所指结点信息域的内容，如此重复下去，直至 p 所指结点的信息域值为 x 或 p=head 为止。

5. 在中序线索二叉树上插入结点

在中序线索二叉树上插入结点可以分为两种情况考虑。一种情况是将新结点插入到二叉树中作为某结点的左儿子结点，另一种情况是将新结点插入到二叉树中作为某结点的右儿子结点，以下仅讨论后一种情况。

假设指针变量 p 指向二叉线索树中的某结点，指针变量 r 指向要插入的新结点，新结点 r 将插入二叉树中，作为结点 p 的右儿子。

若结点 p 的右子树为空，则插入过程很简单，如图 5-16(a)、(b)所示。这时结点 p 原来的后继结点变为结点 r 的后继，结点 p 为结点 r 的前趋，结点 r 成为结点 p 的右儿子。结点 r 插入对于结点 p 原来的后继结点没有任何影响。

若结点 p 的右子树不为空，则结点 r 插入后，结点 p 原来的右子树变为结点 r 的右子树，p 为 r 的前趋结点，r 为 p 的右儿子结点。根据中序线索二叉树的定义，这时还需修改结点 p 原来右子树中最左结点的左指针域，使它由原来的指向结点 p 改为指向结点 r，如图 5-16(c)、(d)所示。

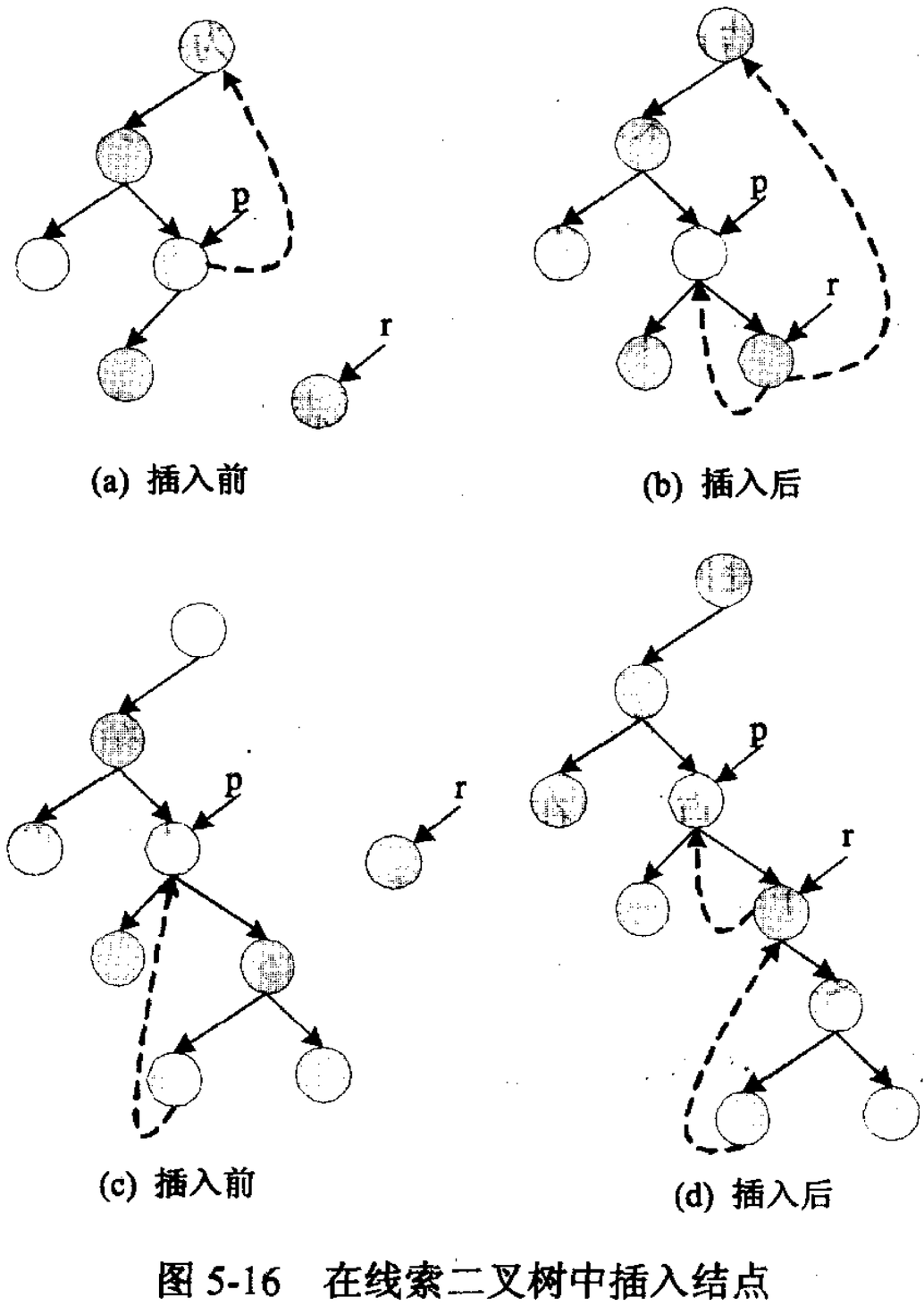


图 5-16　在线索二叉树中插入结点

将新结点插入到线索二叉树中作为某结点的左儿子结点，以及从线索二叉树中删除结点的算法与上述算法的分析设计方法相类似。

从上述算法可看出，往一棵线索二叉树中插入结点或者从一棵线索二叉树中删除结点要作的主要工作就是修改一些指向儿子结点的指针和一些指向前趋结点或后继结点的线索。指向儿子结点的指针的修改很容易，而线索的修改有时花费的代价较大。例如，在上述插入算法中，当 p 有右儿子结点时，插入结点 r 要修改结点 p 的原来后继结点的左线索，这时须首先在结点 p 的右子树中沿左儿子指针不断向下搜索才能找到该后继结点。所以，在对线索二叉树作插入或删除操作时，也可采用另一种方法，插入或删除时只修改指向儿子结点的指针，然后对插入或删除后的二叉树重新进行线索化。

5.7　树和二叉树的转换及树的存储结构

本节讲述的主要内容为树与二叉树的转换规则、森林与二叉树的转换规则、树的遍历以及树的存储结构。

5.7.1　树转换为二叉树

对于一棵无序树，树中结点的各儿子的次序是无关紧要的，而二叉树中结点的左、右儿子结点是有区别的。为避免发生混淆，我们约定树中每一个结点的儿子结点按从左到右的次序顺序编号，也就是说，把树作为有序树看待。如图 5-17 所示的一棵树，根结点 *A* 有三个儿子 *B*、*C*、*D*，可以认为结点 *B* 为 *A* 的第一个儿子结点，结点 *C* 为 *A* 的第二个儿子结点，结点 *D* 为 *A* 的第三个儿子结点。

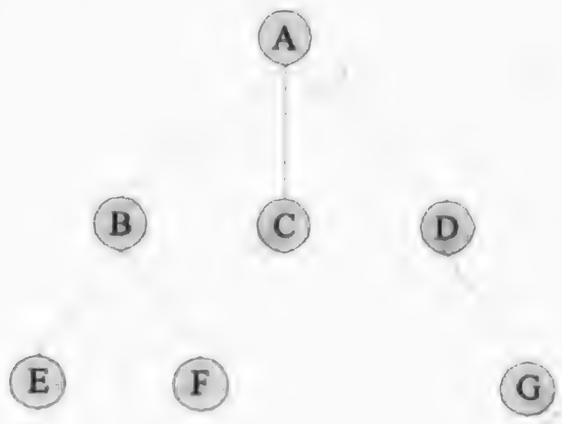


图 5-17　一般树

将一棵树转换为二叉树的方法是：

(1) 树中所有相邻兄弟之间加一条连线；

(2) 对树中的每个结点，只保留它与第一个儿子结点之间的连线，删去它与其他儿子结点之间的连线。

(3) 以树的根结点为轴心，将整棵树顺时针转动一定的角度，使之结构层次分明。

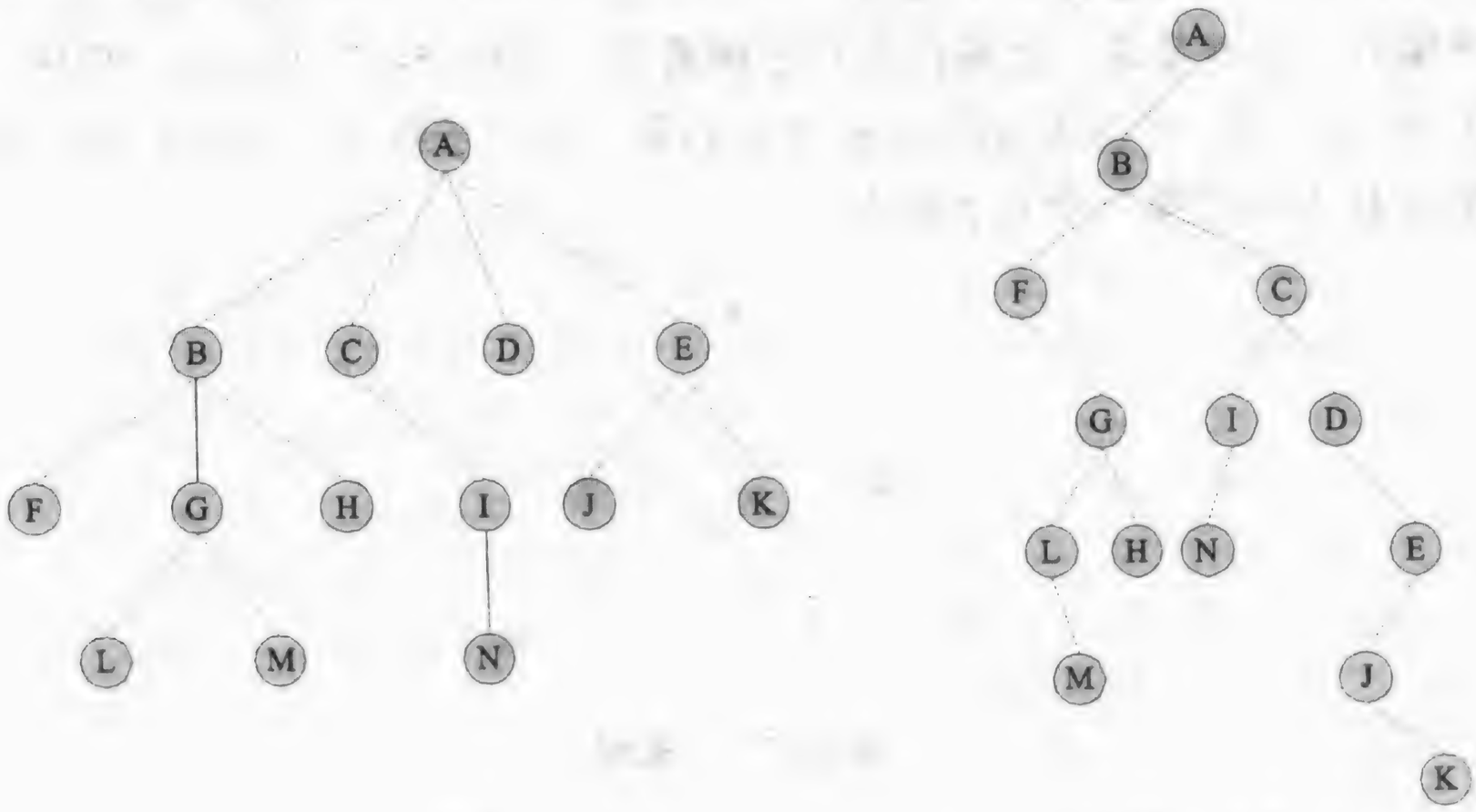
可以证明，树作这样的转换所构成的二叉树是惟一的。图 5-18(a)、(b)、(c)给出了图 5-17 所示的树转换为二叉树的转换过程示意图。

(a) 相邻兄弟加连线　　(b) 删去双亲与其他儿子的连线　　(c) 转换后的二叉树

图 5-18　树转换为二叉树的过程

图 5-19 给出了另一棵树及其转换后所建立的二叉树。通过转换过程可以看出，树中的任意一个结点都对应于二叉树中的一个结点，树中某结点的第一个儿子在二叉树是相应结

点的左儿子，就树中的相邻兄弟结点而言，在二叉树中后一个儿子结点成为前一个儿子结点的右儿子。也就是说，在二叉树中，左分支上的各结点在原来的树中是父子关系，而右分支上的各结点在原来的树中是兄弟关系。由于树的根结点没有兄弟，所以变换后的二叉树的根结点的右孩子始终为空。



(a) 一棵树　　(b) 二叉树

图 5-19　树及其转换后的二叉树

5.7.2　二叉树还原为树

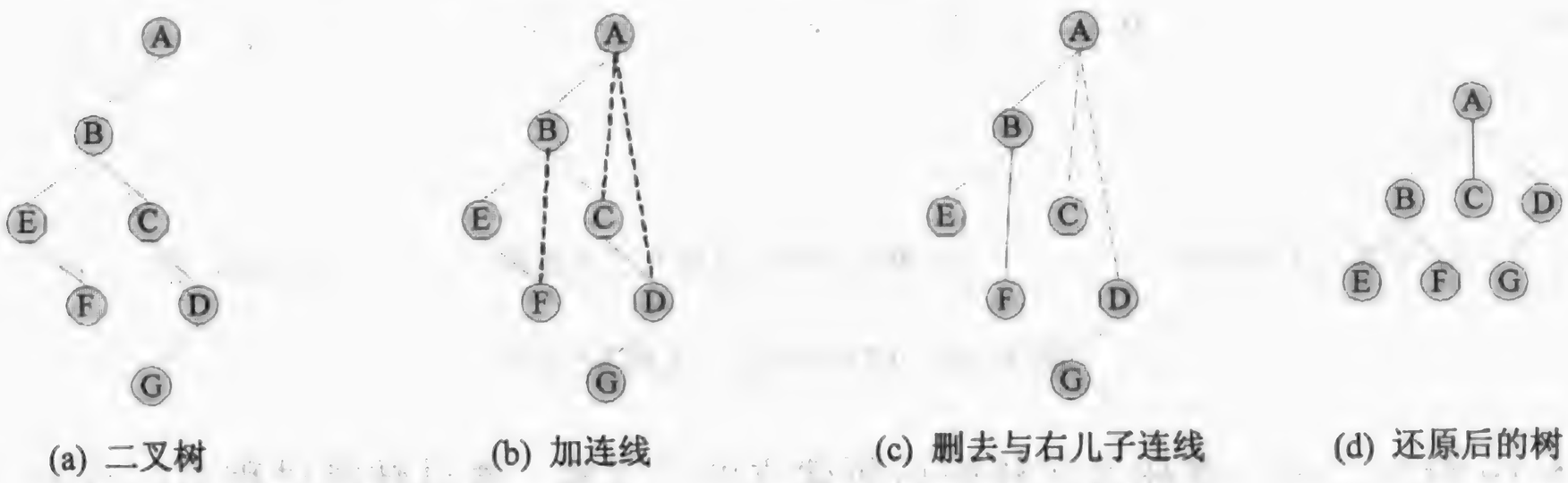
树转换为二叉树这一转换过程是可逆的，可以依据二叉树的根结点有无右儿子结点，将一棵二叉树还原为树，具体方法如下：

(1) 若某结点是其双亲的左儿子，则把该结点的右儿子、右儿子的右儿子、…都与该结点的双亲结点用线连起来；

(2) 删掉原二叉树中所有的双亲结点与右儿子结点的连线；

(3) 整理由(1)、(2)两步所得到的树，使之结构层次分明。

图 5-20 所示为一棵二叉树还原为树的过程示意图。



(a) 二叉树　　(b) 加连线　　(c) 删去与右儿子连线　　(d) 还原后的树

图 5-20　二叉树还原为树的过程

5.7.3　森林转换为二叉树

森林是若干棵树的集合，森林亦可用二叉树表示。

森林转换为二叉树的方法如下：

(1) 将森林中的每棵树转换成相应的二叉树；

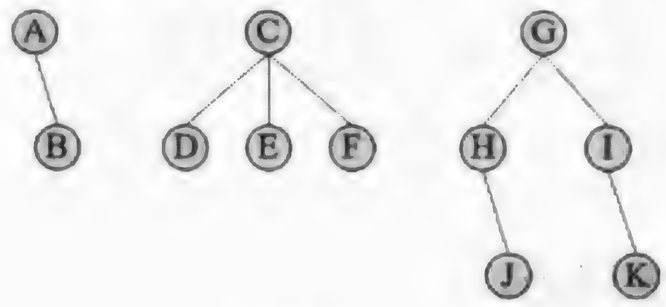
(2) 第一棵二叉树不动，从第二棵二叉树开始，依次将后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有的二叉树连在一起后，这样所得到的二叉树就是由森林转换得到的二叉树。

设 $F=\{T_1,T_2,\cdots,T_n\}$是森林，其所对应的二叉树为 $B(T_1,T_2,\cdots,T_n)$，有：

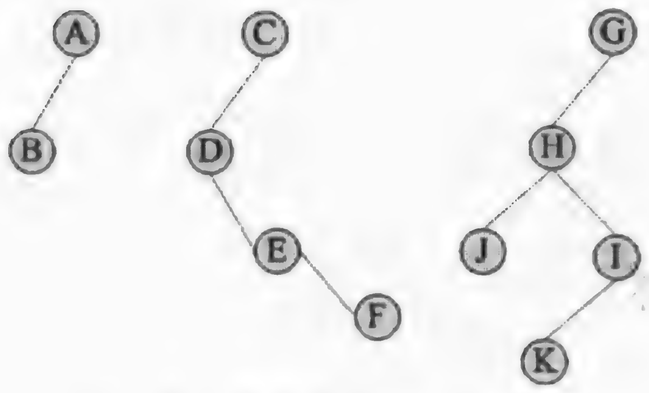
(1) 若 $n=0$，即 F 为空，那么 B 亦为空；

(2) 若 $n>0$，则二叉树的根结点为树 T_1 的根结点，其左子树为 $B(T_{11},T_{12},\cdots,T_{1m})$，其中 T_{11}，T_{12}，…，T_{1m} 是根结点 T_1 的子树，而其右子树为 $B(T_2,T_3,\cdots,T_n)$。

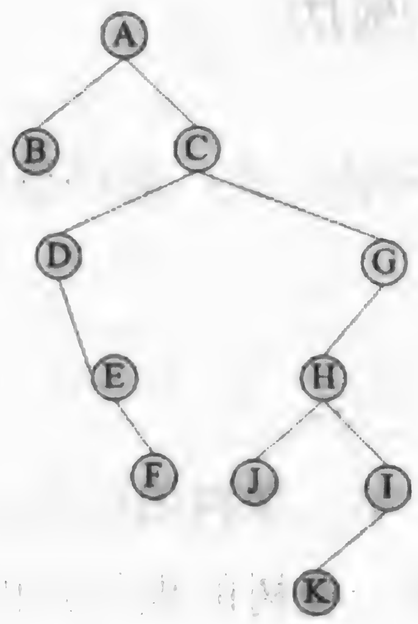
如图 5-21 所示为森林及其转换为二叉树的过程。



(a) 森林



(b) 每一棵树转换为二叉树



(c) 每一棵二叉树连接后的二叉树

图 5-21　森林及其转换为二叉树的过程

5.7.4　树的遍历

树的遍历通常有两种方式：先根遍历和后根遍历。下面分别进行讨论。

1. 先根遍历

先根遍历的定义为：

(1) 访问根结点；

(2) 按照从左到右的顺序先根遍历根结点的每一棵子树。

按照树的先根遍历的定义，对图 5-17 所示的树进行先根遍历，得到的结果序列为 *A B E F C D G*。

同样，对图 5-19(a)所示的树进行先根遍历，得到的结果序列为 *A B F G L M H C I N D E J K*。

2. 后根遍历

后根遍历的定义为：

(1) 按照从左到右的顺序后根遍历根结点的每一棵子树；

(2) 访问根结点。

按照树的后根遍历的定义，对图 5-17 和图 5-19(a)所示的树进行后根遍历，得到的结果序列分别为 *E F B C G D A*、*F L M G H B N I C D J K E* A

事实上，根据树与二叉树的转换关系以及树和二叉树的遍历定义可以推知，树的先根遍历与其转换的相应二叉树的前序遍历的结果序列相同；树的后根遍历与其转换的相应二叉树的中序遍历的结果序列相同。因此，树的遍历算法也可采用相应的二叉树的遍历算法实现。

5.7.5　森林的遍历

森林的遍历有两种方式：前序遍历和中序遍历。

1. 前序遍历

前序遍历的定义为：

(1) 访问森林中第一棵树的根结点；

(2) 前序遍历第一棵树的根结点的子树森林；

(3) 前序遍历剩余的其他子森林。

对于图 5-21 所示的森林进行前序遍历，得到的结果序列为 *A B C D E F G H I J K*。

2. 中序遍历

中序遍历的定义为：

(1) 中序遍历第一棵树的根结点的子树森林；

(2) 访问森林中第一棵树的根结点；

(3) 中序遍历剩余的其他子森林。

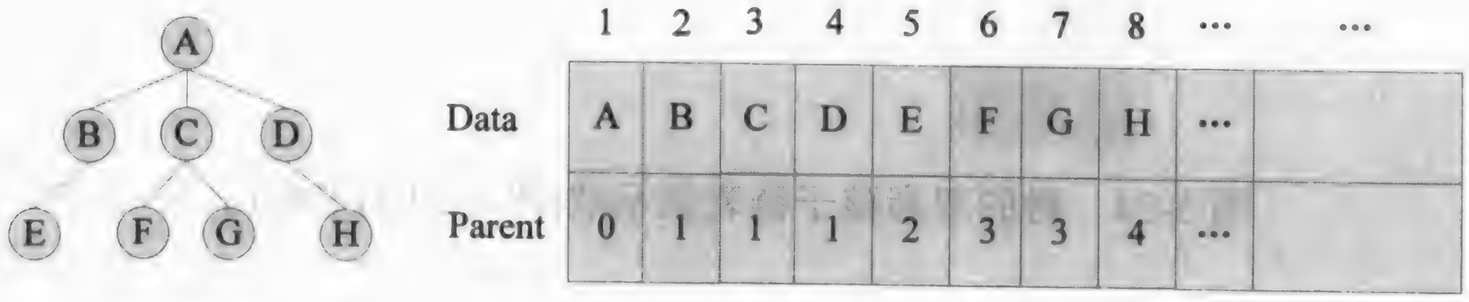
对于图 5-21 所示的森林进行中序遍历，得到的结果序列为 *B A D E F C J H K I G*。

根据森林与二叉树的转换关系以及森林和二叉树的遍历定义可以推论：森林前序遍历和中序遍历分别与所转换的二叉树的前序遍历和中序遍历的结果序列相同。

5.7.6 树的存储结构

1. 双亲链表表示法

以一组连续的存储单元来存放树中的结点，每个结点有两个域：一个是数据域，用来存放结点信息，另一个是双亲域，用来存放双亲的位置(指针)。该结构的具体描述如图 5-22 所示。



(a) 树　　(b) 树的双亲链表表示法

图 5-22 树的双亲链表表示法示意

2. 孩子链表表示法

将一个结点所有孩子链接成一个单链表形，而树中有若干个结点，则有若干个单链表，每个单链表有一个表头结点，所有表头结点用一个数组来描述，具体描述如图 5-23 所示。

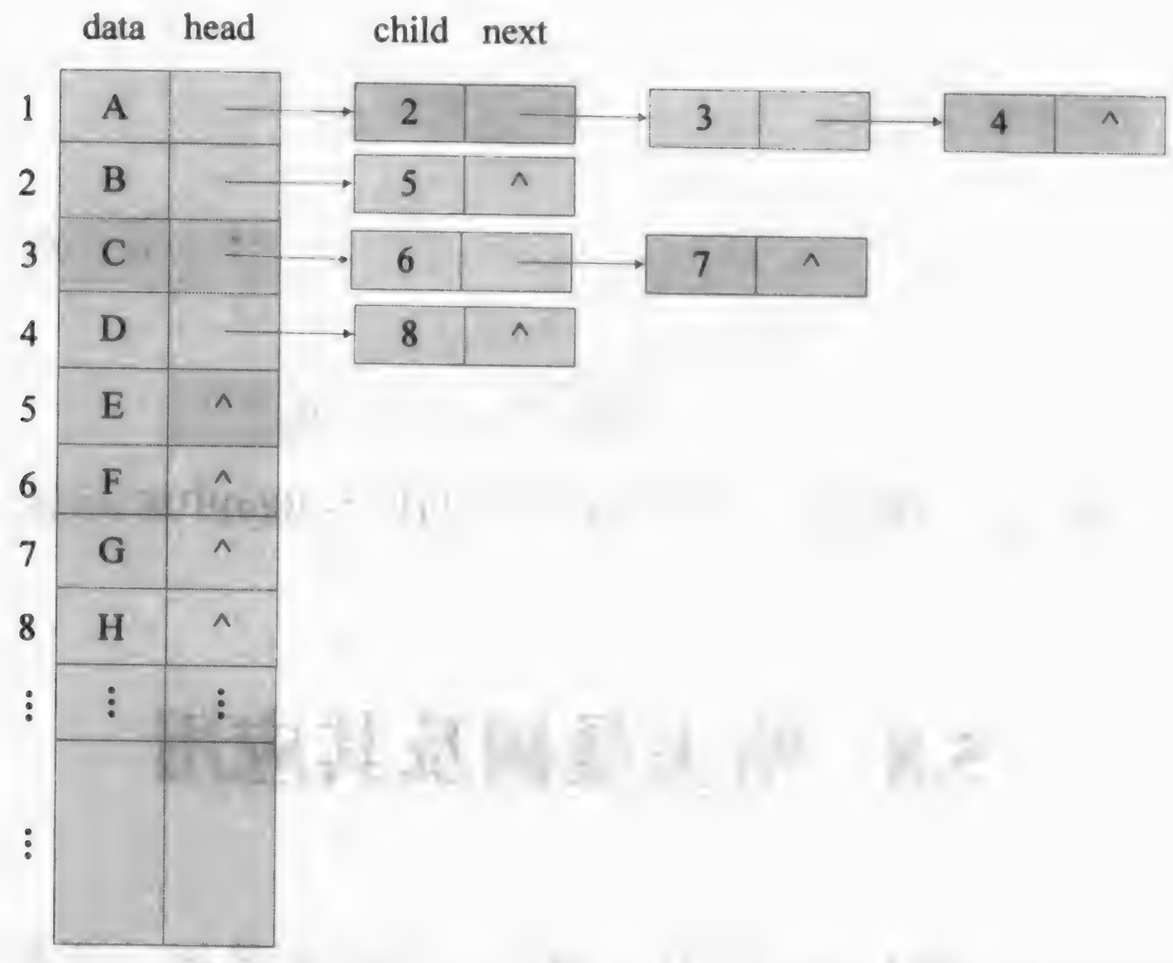


图 5-23 树的孩子链表表示法(图 5-20(a)中树)示意

3. 双亲孩子链表表示法

双亲孩子链表表示法的具体描述如图 5-24 所示。

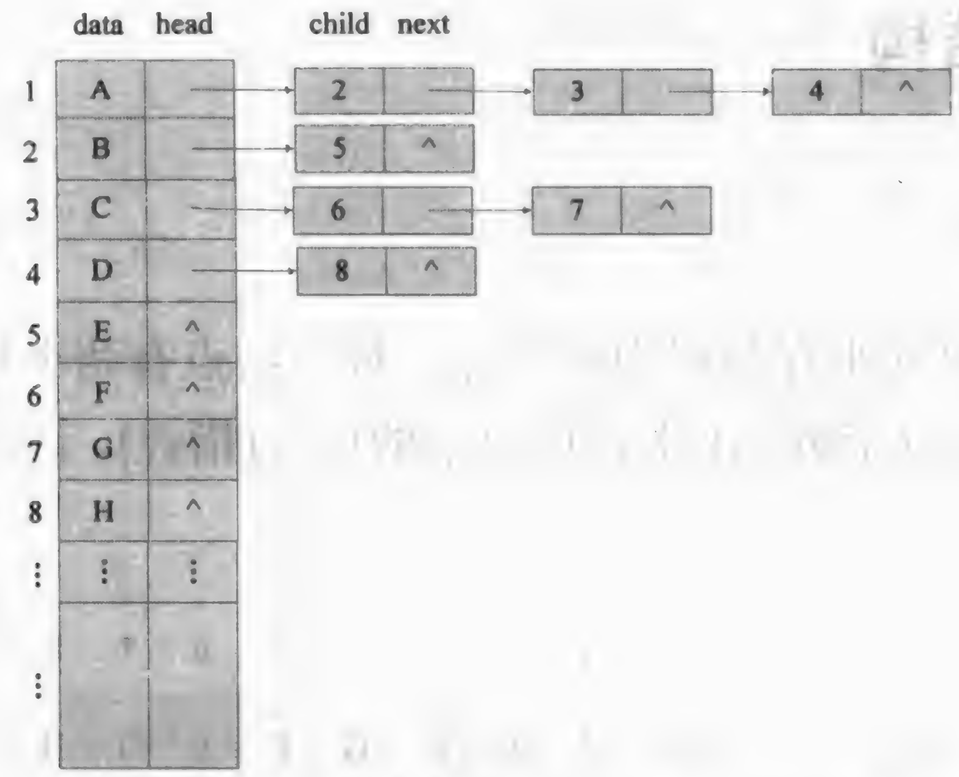


图 5-24　树的双亲孩子链表表示法(图 5-20(a)中树)示意

4. 孩子兄弟链表表示法

类似于二叉链表，但第一根链指向第一个孩子，第二根链指向下一个兄弟。将图 5-21(a)的树用孩子兄弟链表表示法表示，如图 5-25 所示。

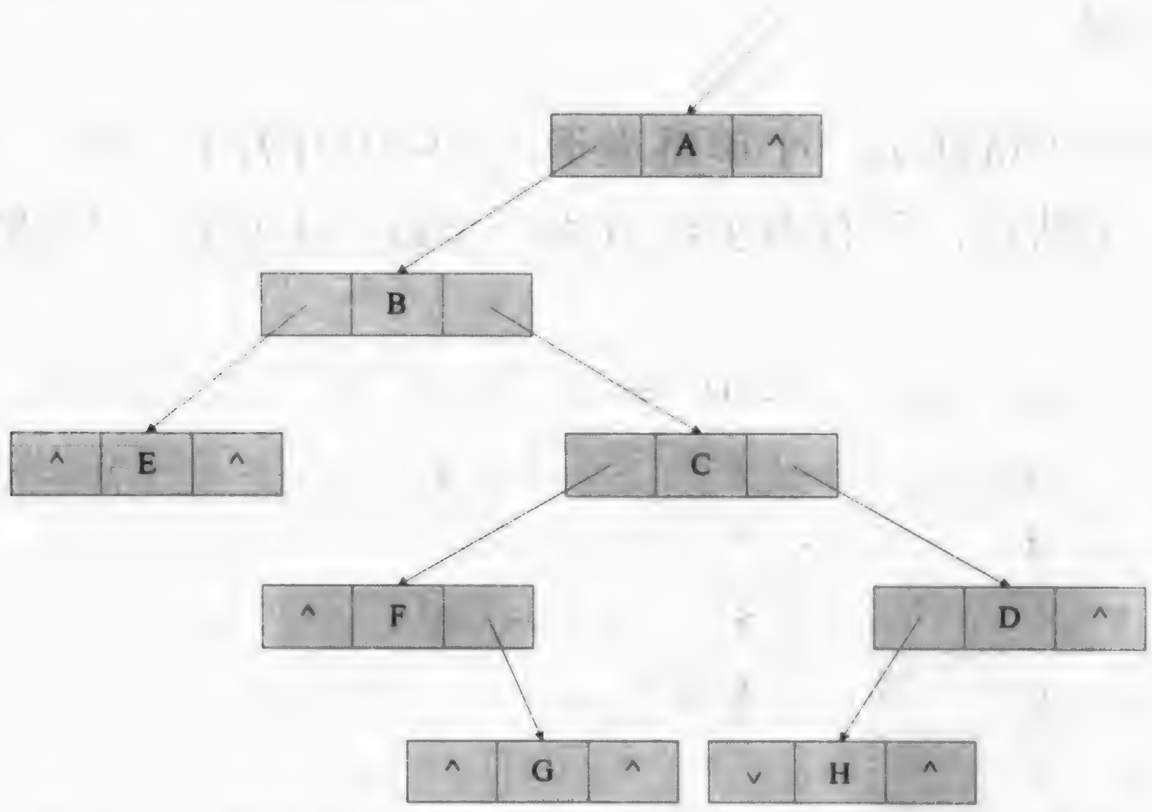


图 5-25　树的孩子兄弟链表表示法(图 5-20(a)中树)示意

5.8　哈夫曼树及其应用

本节讲述的主要内容为哈夫曼树的基本概念以及哈夫曼树的两种应用：编码问题和判定问题。

5.8.1　哈夫曼树的基本概念

1. 路径和路径长度

在一棵树中，从一个结点往下可以达到的孩子或子孙结点之间的通路，称为路径。通路中分支的数目称为路径长度。

若规定根结点的层数为 1，则从根结点到第 L 层结点的路径长度为 $L-1$。

2. 结点的权及带权路径长度

若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。

结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。

3. 树的带权路径长度

树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为 WPL。

4. 最优二叉树

给定 n 个权值作为 n 个叶子结点，按一定规则构造一棵二叉树，使带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树。

5. 哈夫曼树的构造

假设有 n 个权值，则构造出的哈夫曼树有 n 个叶子结点。n 个权值分别设为 W_1、W_2、…、W_n，则哈夫曼树的构造规则为：

(1) 将 W_1、W_2、…、W_n 看成是有 n 棵树的森林(每棵树仅有一个结点)；

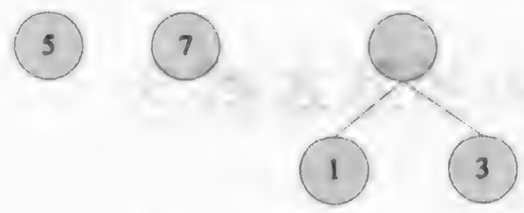
(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3) 从森林中删除选取的两棵树，并将新树加入森林；

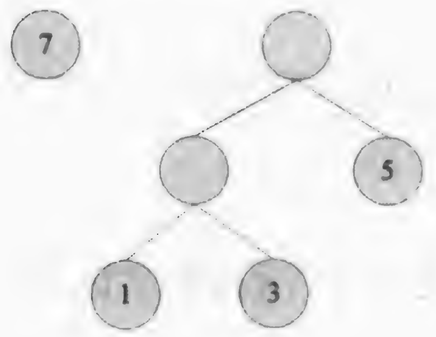
(4) 重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为我们所求得的哈夫曼树。

哈夫曼树的构造过程示意如下：假设给定的叶子结点的权分别为 1、5、7、3，则构造哈夫曼树的过程如图 5-26 所示。从图 5-26 可知，n 个权值构造哈夫曼树需 $n-1$ 次合并，每次合并，森林中的树数目减 1，最后森林中只剩下一棵树，即为求得的哈夫曼树。

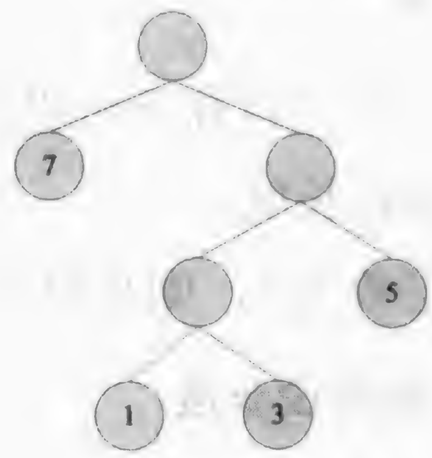
(a) 初始森林



(b) 一次合并后的森林



(c) 二次合并后的森林



(d) 三次合并后的森林

图 5-26　哈夫曼树的构造过程

哈夫曼树类的声明如下：

```
class LettFreq{ //叶结点与权值对
   private char lett; //叶结点
   private int freq; //权值

   public LettFreq(int f, char l) {freq=f; lett=l;}
   public LettFreq(int f) {freq=f;}
   public int weight() {return freq;l}//返回权值
   public char letter() {return lett;}//返回叶结点
}//class LettFreq

class HuffTree{ //哈夫曼编码树
   private BinNode root; //哈夫曼编码树根结点
   public HuffTree(LettFreq val)
{root = new BinNodePtr(val);}
public HuffTree(LettFreq val, HuffTree l, HuffTree r)
{root = new BinNodePtr(val, l.root(), r.root());}

public BinNode root(){ return root;}
public int weight() //根结点的权值即是树的权值
{ return ((LettFreq)root.element()).weight(); }
}//class HuffTree
```

5.8.2　哈夫曼树在编码问题中的应用

在数据通信中，经常需要将传送的文字转换成由二进制字符 0、1 组成的二进制串，我们称之为二进制编码。在发送端，需要将电文中的字符转换成二进制的 0、1 序列(编码)，而在接收端则要将收到的 0、1 序列转换成对应的字符序列(译码)。

最简单的编码方式是等长编码，例如如果电文是英文，则电文的字符串由 26 个英文字母组成，需要编码的字符集合是{A,B,…,Z}。采用等长的二进制编码时，每个字符用五位二进制位串表示即可(2^5>26)。在接收端，只要按五位分割进行译码就可得到对应的字符。

哈夫曼树可用于构造使电文的编码总长最短的编码方案。具体作法如下：设需要编码的字符集合为{$d_1,d_2,\cdots,d_n$}，而它们在电文中出现的次数或频率集合为{$w_1,w_2,\cdots,w_n$}，以$d_1,d_2,\cdots,d_n$作为叶结点，$w_1,w_2,\cdots,w_n$作为它们的权值构造一棵哈夫曼树，规定哈夫曼树中的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径分支组成的 0 或 1 序列便为该结点对应字符的编码，称之为哈夫曼编码。

在哈夫曼编码树中，树的带权路径长度的含义是各个字符的码长与其出现次数的乘积和，也就是电文的代码总长。显然，因为哈夫曼算法构造的是带权路径长度最小的二叉树，所以采用哈夫曼树构造的编码是一种能使电文代码总长最短的不等长编码。

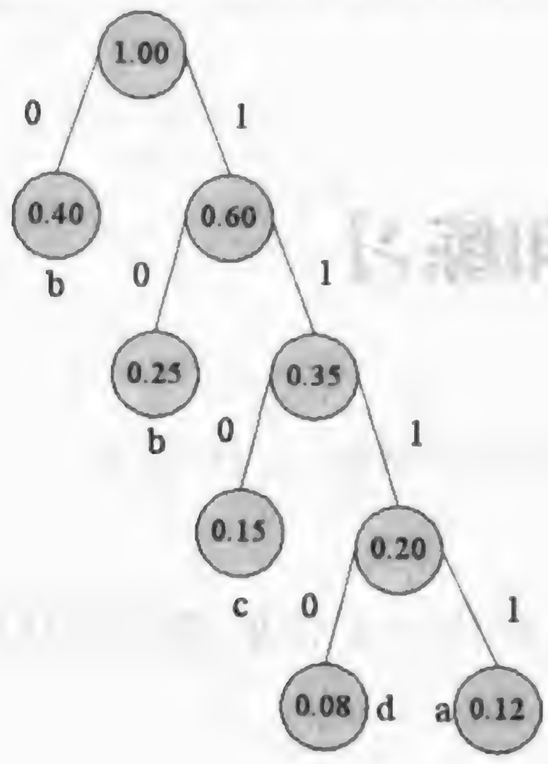
在建立不等长编码时，必须使任何一个字符的编码都不是另一个字符编码的前缀，这样才能保证译码的惟一性。例如，若字符 A 的编码为 0l，字符 B 的编码为 010，那么字符 A 的编码就成了字符 B 的编码的前缀，这时对于代码串 01010……，在译码时就无法判定是将前两位码 0l 译为字符 A，还是将前三位码 010 译为字符 B。在哈夫曼树中，由于每个字符结点都是叶结点，它们不可能在根结点到其他字符结点的路径上，所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀，从而保证了译码的非二义性。

例如，设组成电文的字符集 D 及其概率分布 W 为：

D={a,b,c,d,e}

W={0.12,0.40,0.15,0.08,0.25}

用哈夫曼算法构造的哈夫曼树及其对应的哈夫曼编码如图 5-27 所示。



(a) 哈夫曼编码树

a: 1 1 1 1

b: 0

c: 1 1 0

d: 1 1 1 0

e: 1 0

(b) 哈夫曼编码

图 5-27 哈夫曼编码树及其对应的哈夫曼编码

下面讨论实现哈夫曼编码的算法。

实现哈夫曼编码的算法可分为两大部分，构造哈夫曼树和在哈夫曼树上求叶结点的编码。在构造哈夫曼树时，可以设置一个结构数组 huff_node 保存哈夫曼树中各结点的信息，

根据二叉树的性质可知，具有 n 个叶结点的哈夫曼树共有 $2n-1$ 个结点，所以数组 huff_node 的大小设置为 $2n-1$，数组元素的结构形式如下：

weight	lchild	rchild	flag	parent

其中 weight 域保存结点的权值，lchild 和 rchild 域分别保存该结点的左、右孩子结点在数组 huff_node 中的序号，从而建立起结点之间的关系。为了判定一个结点是否已加入了要建立的哈夫曼树中，设置了一个标志域 flag，当 flag=0 时，表示该结点未加入树中，当 flag=1 时，则表示该结点已加入树中。

在求各字符的编码时，要从叶结点回退到根结点，因此需设置一个 parent 域，用来保存结点的双亲结点在 huff_node 数组中的序号。

构造哈夫曼树时，首先将由 n 个字符形成的 n 个叶结点存放到数组 huff_node 的前 n 个分量中，然后根据哈夫曼算法的基本思想，不断将小子树合并为较大的子树，每次所构成的新子树的根结点顺序放到 huff_node 数组中的前 n 个分量的后面。

求哈夫曼编码，实质上就是从叶结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树中的一个分支，从而得到一位哈夫曼码值，由于一个字符的哈夫曼编码是从根结点到相应叶结点所经过的路径上各分支组成的 0、1 序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为所求编码的高位码。可以设置一结构数组 huff_code 用来存放各字符的哈夫曼编码信息，数组元素的形式如下：

bits	Start

其中分量 bits 为一维数组，用来保存字符的哈夫曼编码，start 表示该编码在数组 bits 中的开始位置，所以对于第 i 个字符，它的哈夫曼编码存放在 huff_code[i].bits 中的从 huff_code.start 到 n 的分量上。

思考和练习

(1) 若一棵树中度为 1 的结点有 n_1 个，度为 2 的结点有 n_2 个，……，度为 m 的结点有 n_m 个，它有多少个叶结点？

(2) 找出所有的二叉树，其结点在下列两种次序下恰好都以同样的顺序出现：

- 先根和中根
- 先根和后根
- 中根和后根

(3) 设计一个算法，根据一个二叉树结点的先根序列和中根序列构造出该二叉树。假设二叉树是链接表示的，并且任意两个结点的数据字段值都不同。

(4) 设计一个算法，将一个链接表示的二叉树中每个结点的左、右儿子位置交换。

(5) 设计一个算法，按层次顺序输出二叉树中的所有结点，要求同一层上的结点从左到右输出。

(6) 设 F 是一个森林，B 是与 F 对应的二叉树。试问，F 中非叶结点的个数和 B 中右子树为空的结点的个数之间有什么数量关系？

(7) 将图 5-28 所示树转换为二叉树。

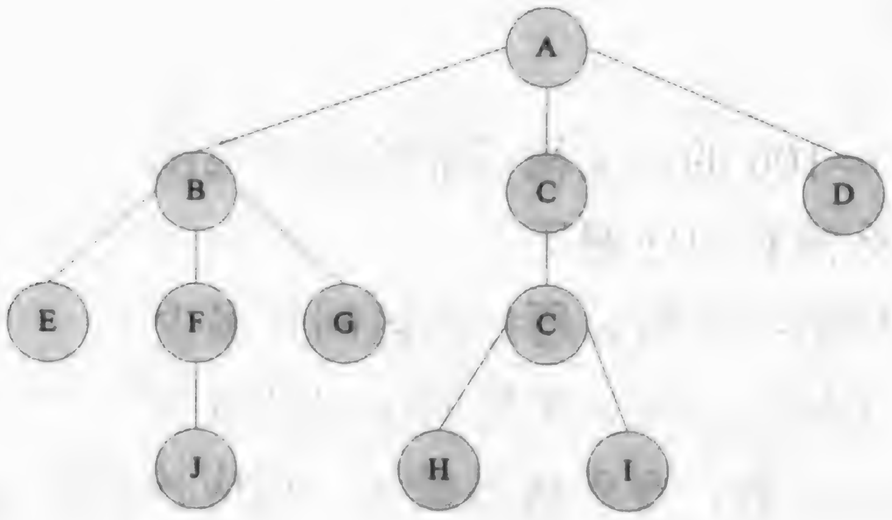


图 5-28　题 7 图

(8) 写一个列出二叉树中所有叶结点的函数。

(9) 写一个递归函数找出二叉树中最长路径的长度。

(10) 写一个计算二叉树中所有结点个数的函数。

(11) 判断真假：

- 树中结点的深度等于它的祖先的个数。
- 子树的大小等于该子树的根结点的后代数。
- 如果 x 是 y 的后代，则 x 的深度大于 y 的深度。
- 如果 x 的深度大于 y 的深度，则 x 是 y 的后代。
- 一棵树是单结点树，当且仅当它的根是一个叶子结点。
- 一个结点的祖先数等于它的深度。
- 如果 R 是 S 的子树，S 是 T 的子树，则 R 是 T 的子树。
- 一个结点是叶子结点当且仅当它的度是 0。
- 在任意一棵树中，内部结点的数目一定少于叶子结点的数目。
- 一棵树是满的，当且仅当它的叶子结点位于相同的层上。
- 满二叉树的每棵子树也是满树。
- 满二叉树的每棵子树也是完全树。
- 如果一棵二叉树的所有叶子结点都处于同层，这棵二叉树是满二叉树。
- 如果二叉树的结点数为 n，高度为 h，则有 $h \geqslant \lfloor \log_2^n \rfloor$。
- 二叉树的深度为 d，则其最多有 $2d$ 个结点。
- 如果一棵二叉树的每个合法的子树都是满二叉树，则这棵二叉树本身是满二叉树。

(12) 哪一种遍历方式总是：

- 首先访问根结点？

- 位于最左边的结点最先访问？
- 根最后访问？
- 最右边的结点最后访问？

(13) 一棵高度为 9 的满二叉树：

- 有多少个叶子结点？
- 有多少个中间结点？
- 有多少个结点？

(14) 给出一棵结点数 n=100 的二叉树可能的高度范围。

(15) 对于一般树为什么没有中序遍历？

(16) 高度为 4 的二叉树的结点数 n 所处的区间是多少？

(17) 结点数为 7 的二叉树的高度 h 所处的区间是多少？

(18) 给定的二叉树的结点数，要使树高最大，树应该是什么形状？要使树高最小，树又应该是什么形状？

(19) 试画出大小 n=5 的所有 42 棵二叉树。

(20) 大小 n=6 的不同二叉树有多少棵？

(21) 如果用 $f(n)$表示大小为 n 的二叉树的数目，试推导出 $f(n)$的循环关系。

(22) 给出高度为 h 的完全二叉树的数目 $f(h)$的计算公式。

(23) 给出高度为 h 的满二叉树的数目 $f(h)$的计算公式。

(24) 证明满二叉树的每棵子树也都是满二叉树。

(25) 证明完全二叉树的每棵子树也都是完全二叉树。

第6章　图

图是非线性数据结构。图中任何两个顶点都可能有关联，顶点间的关系是多对多的关系，这种关系在现实世界中大量存在。

本章的学习目标：

- 图的概念；
- 图的存储结构；
- 图的遍历；
- 生成树和最小生成树；
- 最短路径和拓扑排序。

6.1　图的基本概念

本节讲述的主要内容为图的定义以及图的常用术语。

6.1.1　图的定义

1. 图的定义

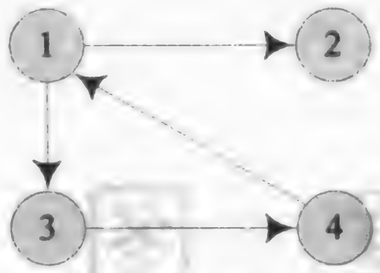
图可用二元组表示：$G=(V,E)$，其中 V 表示元素(顶点)的非空有限集合；E 表示元素之间关系(边)的有限集集合，边是顶点偶对。可以看出图的每个结点有任意多个前趋和后继结点。

2. 有向图和无向图

如果图的边限定为从一个顶点指向另一个顶点，即每条边都是顶点的有序偶对，称之为有向图。如果图中的边没有方向性，即每条边都是顶点的无序偶对，称之为无向图。如图 6-1 表示一个有向图和无向图。

这里，有向图中<1,2>和<2,1>表示两个不同的方向边。以<1,2>为例，在<1,2>中：1 称为此边的起点或尾(孤尾)，2 称为此边的终点或头(孤头)。边的方向规定为孤尾⟶孤头。

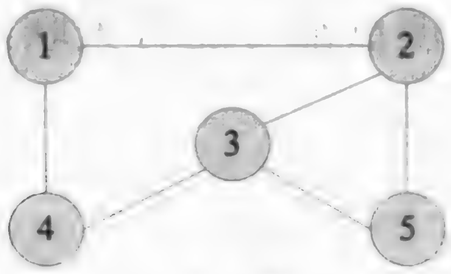
而无向图中(1,2)和(2,1)代表同一边。



(a) 有向图 *G*1

V(*G*1)={1,2,3,4}顶点集合；

E(*G*1)={<1,2>,<1,3>,<3,4>,<4,1>}边的集合



(b) 无向图 *G*2

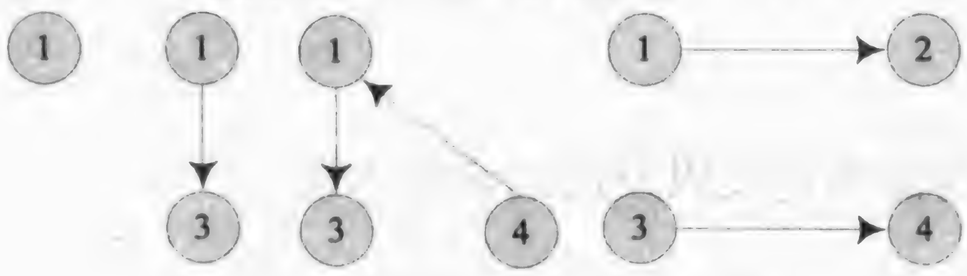
V(*G*2)={1,2,3,4,5}顶点集合；

E(*G*2)={(1,2)，(1,4)，(2,3)，(2,5)，(3,4)，(3,5)}边的集合

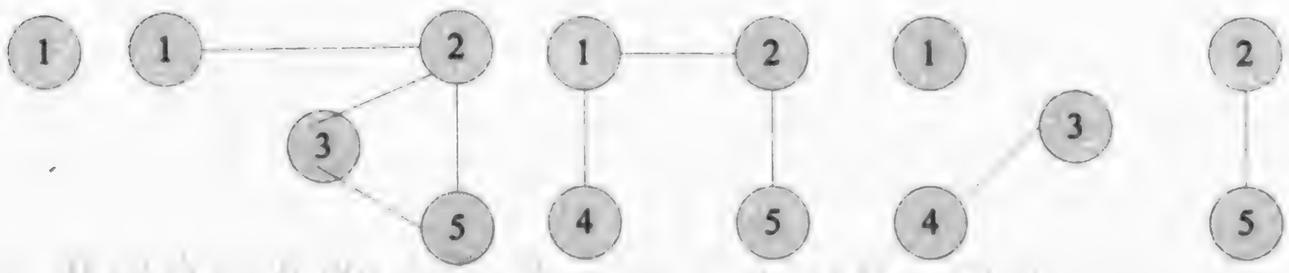
图 6-1　有向图和无向图

3. 子图

若 *G* 和 *G*′是两个图，且存在着 $V(G')\subseteq V(G)$和 $E(G')\subseteq E(G)$的关系，则称 *G*′是 *G* 的子图。如图 6-2 表示图与子图。



(a) 图 6-1 中有向图 *G*1 的子图示例



(b) 图 6-1 中无向图 *G*2 的子图示例

图 6-2　有向图、无向图及其子图

6.1.2　常用术语

1. 完全图

在一个有 *n* 个顶点的无向图中，若每个顶点到其他 *n* - 1 顶点都有一条边，则图中有 *n* 个顶点且有 *n**(*n* - 1)/2 条边的图称为无向完全图。若为有向完全图，则边应为 *n**(*n* - 1)。可见在无向完全图中，任意两点之间均有边；在有向完全图中，任意两顶点之间均有方向相反的两条弧。

图 6-3 中，顶点数=4，边数=4*(4 - 1)/2=6。可以看到，图中每个顶点到其他三个顶点都有一条边相连。

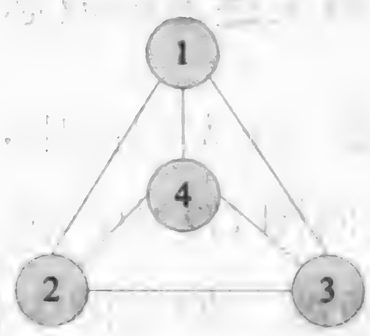


图 6-3　无向完全图

2. 邻接点

对无向图 $G=(V,E)$，若有$(V1,V2)\in E$，则称 $V1$ 和 $V2$ 互为邻接点。如图 6-3 中，结点 4 的邻接点有 1、2、3，称 1 和 4、2 和 4 互为邻接点。

3. 相关边

两个相邻接的点连成的边叫做这两个结点的相关边。如图 6-3 中，与结点 4 相关联边有(1,4)、(2,4)和(3,4)。

4. 度

与每个顶点相连的边的数叫该点的度。如图 6-3 中，顶点 4 的度为 3，其他各点度也均为 3。

5. 入度

对有向图中某结点的孤头数(边的终点)称为该结点的入度。如图 6-1(a)有向图 $G1$ 中，结点 4 的入度为 1，因为只有结点 3 指向它。

6. 出度

对有向图中某结点的孤尾数(边的终点)称为该结点的出度。对于有向图，有度=入度+出度。如图 6-1(a)有向图 $G1$ 中，结点 4 的出度为 1，因为它只指向结点 1。

7. 路径

在一图中若从某个顶点 V_P 出发，沿着一些边经过顶点 V_1，V_2，…，V_m 到达 V_g 则称顶点序列$(V_P,V_1,V_2,\cdots,V_m,V_g)$为从 V_P 到 V_g 的路径，其中 V_P 是路径的始点，V_g 是路径的终点。如图 6-1(b)无向图 $G2$ 中，从 1 到 5 的路径为 1、4、3、5 或 1、2、5。

对于有向图路径也是有向的，路径的方向总从起点到终点，且与它经过的每条边的方向一致。

路径上的边或弧的数目称之为该路径的长度。

除始点和终点外，其他各顶点均不同的路径称之为简单路径。

8. 回路

从一顶点出发又回到该顶点，则所经过的路径(即始点和终点相同的路径)称为回路。

始点和终点相同的简单路径称之为简单回路。

9. 有关连通的概念

连通：在无向图中，若从顶点 V_i 到顶点 V_j 之间有路径则称这两个顶点是连通的。如图 6-1(b)无向图 *G*2 中，顶点 1 到顶点 5 是有路径的，所以是连通的。

连通图：如果图中任意一对顶点之间都是连通的，则称此图为连通图。如图 6-1(b)无向图 *G*2 中，因为任意两点均连通，所以 *G*2 是一个连通图。

连通分量：非连通图中的每一个连通部分叫连通分量。无向图及其 3 个连通分量如图 6-4 所示。

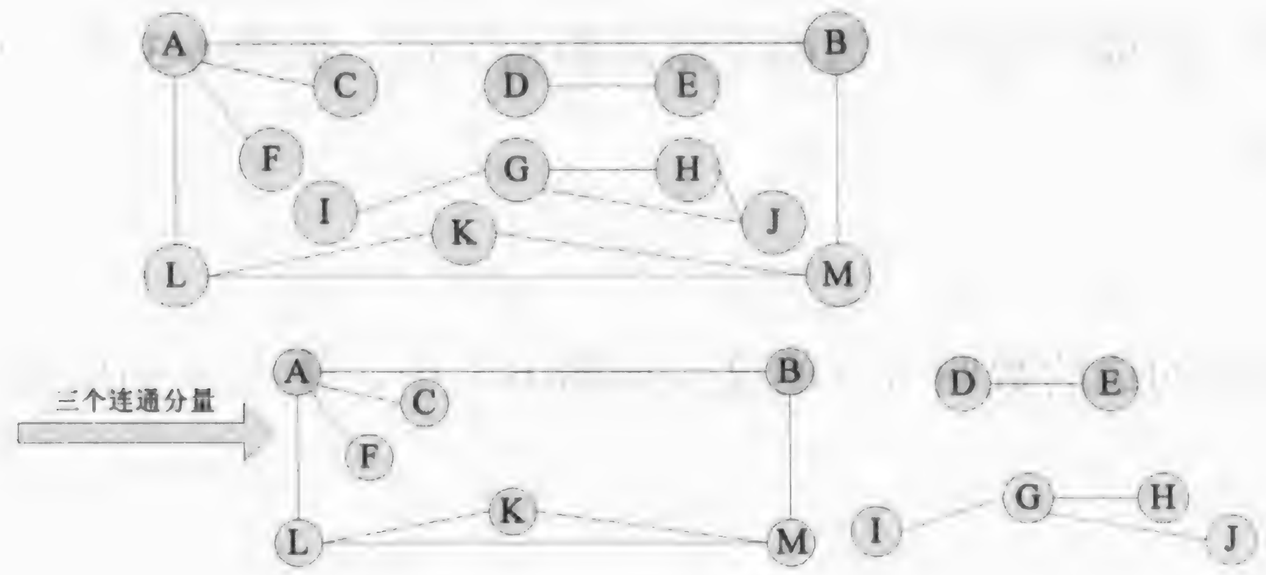


图 6-4　无向图及其三个连通分量

强连通：对于有向图，若从顶点 V_i 到顶点 V_j 到顶点 V_i 之间都有路径，则称这两点是强连通的。如图 6-1(a)有向图 *G*1 中，顶点 1 和顶点 3 是强连通的。

强连通图：如果有向图中任何一对顶点都是强连通的，则此图叫强连通图。如图 6-5 所示，此图不是强连通图，因为顶点 2 到其他顶点不存在路径。

强连通分量：有向图中最大连通子图称为有向图的强连通分量。如图 6-5 为图 6-1(a)有向图 *G*1 的两个强连通分量。

10. 权和带权图(网或者网络)

权：有些图对应每条边有一相应的数值，这个数值称为该边的权。

网：带权的图称为网。网可分为有向网和无向网。

不同网络的权有不同的意义：电网络权可以是阻抗，运输网络中的权可以是路程长度、运费等。带权图如图 6-6 所示。

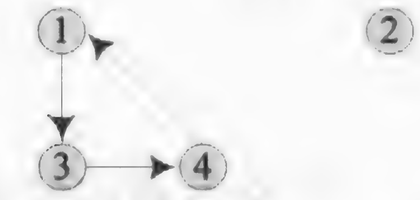


图 6-5　强连通分量

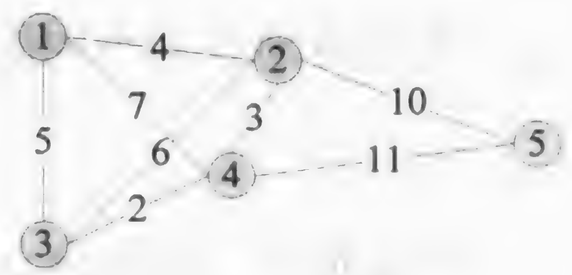


图 6-6　网络(带权图)
(每条边上的数字就是该边的权)

6.2　图的存储结构

本节讲述的主要内容为图的邻接矩阵表示法和邻接表表示法。

为了便于计算机处理图的问题，需要把图的各顶点间的连接关系输入到计算机。只有采用计算机容易接受和处理的数据结构来表示图，才能有利于计算机进行运算。

对具体的图而言，最好的存储结构不仅依赖于数据的性质，而且也依赖于在这些数据上所实施的操作。恰当地选择存储结构也受到其他一些因素的影响。例如：顶点的数目、度的平均数、有向图还是无向图等。

6.2.1　邻接矩阵表示法

邻接矩阵是用来表示图中顶点之间的邻接关系的矩阵。邻接矩阵是：

设 $G=(V,E)$是具有 n 个顶点的图，则 G 的邻接矩阵是一个 $n*n$ 的方阵，其中矩阵每一行分别对应图的各个顶点；矩阵的每一列分别对应图的各个顶点。

规定矩阵元素

$$a_{ij}=\begin{cases}1 & \text{对无向图存在}(v_i,v_j)\text{边,对有向图存在}<v_i,v_j>\text{边}\\ 0 & \text{不存在边}\end{cases}$$

对于无向图，其邻接矩阵是对称的，即 $a_{ij}=a_{ji}$。图 6-1 中图 $G1$ 和 $G2$ 的邻接矩阵分别为：

$$A=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}\qquad B=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

若图的各边是带权的，即图为网络时，用边(弧)的权值 w_{ij} 作为邻接矩阵对应元素的值，定义如下：

$$a_{ij}=\begin{cases}w_{ij} & \text{对无向图存在}(v_i,v_j)\text{边，对有向图存在}<v_i,v_j>\text{边}\\ \infty\text{或者}0 & \text{不存在边}\end{cases}$$

对于无向网，其邻接矩阵是对称的，即 $a_{ij}=a_{ji}$。图 6-6 所示无向网络的邻接矩阵为：

$$C=\begin{bmatrix}0 & 4 & 5 & 7 & 0\\ 4 & 0 & 6 & 3 & 10\\ 5 & 6 & 0 & 2 & 0\\ 7 & 3 & 2 & 0 & 11\\ 0 & 10 & 0 & 11 & 0\end{bmatrix}$$

邻接矩阵的性质：

(1) 图中各顶点序号确定后，图的邻接矩阵是惟一确定的；

(2) 无向图和无向网的邻接矩阵是一个对称矩阵；

(3) 无向图邻接矩阵中第 i 行(或第 i 列)的非 0 元素的个数即为第 i 个顶点的度；

(4) 有向图邻接矩阵第 i 行非 0 元素个数为第 i 个顶点的出度，第 i 列非 0 元素个数为第 i 个顶点的入度，第 i 个顶点的度为第 i 行与第 i 列非 0 元素个数之和；

(5) 无向图的边数等于邻接矩阵中非 0 元素个数之和的一半，有向图的弧数等于邻接矩阵中非 0 元素个数之和。

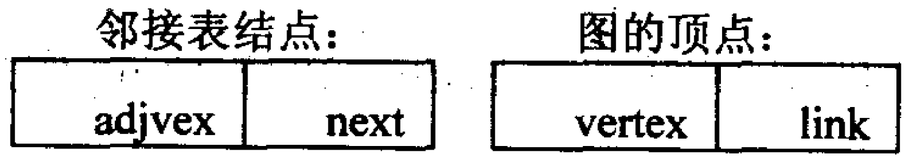
需要说明的是：

(1) 邻接矩阵表示法对于以图的顶点为主的运算比较适用；

(2) 除完全图外，其他图的邻接矩阵有许多零元素，特别是当 n 值较大，而边数相对完全图的边 $n-1$ 又少得多时，则此矩阵称为“稀疏矩阵”，其浪费存储空间。

6.2.2　邻接表表示法

邻接表是图的一种链接存储结构。在邻接表表示法中，用一个顺序存储区来存储图中各顶点的数据，并对图中每上顶点 v_i 建立一个单链表(此单链表称之为 v_i 的邻接表)，把顶点 v_i 的所有相邻顶点，即其后继顶点的序号链接起来。邻接表中的每一个结点(边表结点)均包含有两个域：邻接点域和指针域。邻接点域用于存放与顶点 v_i 相邻接的一个顶点的序号，指针域用于指向下一个边表结点。每个顶点 v_i 除设置存储本身数据外，还设置了指针域，作为邻接表的表头指针。n 个顶点用一维数组表示。



在无向图的邻接表中，顶点 v_i 的每一个边表结点对应于与 v_i 相关联的一条边；而在有向图的邻接表中，v_i 的每一个边表结点对应于以 v_i 为始点的一条弧，因此也称有向图的邻接表的边表为出边表。这样，在有向图的邻接表中求第 i 个顶点的出度很方便，即为第 i 个出边表中结点的个数，但是如果要求求第 i 个顶点的入度则必须遍历整个表。考虑在有向图的邻接表中，将顶点 v_i 的每个边表结点对应于以 v_i 为终点的一条弧，即用边表结点的邻接点域存储邻接到 v_i 的顶点的序号，由此构成的邻接表称为有向图的逆邻接表，逆邻接表有边表称为入边表。这样在逆邻接表中求某顶点的入度与在邻接表中求顶点的出度一样

方便。

对于网络，则只需要在以上边表结点的结构中增设一权值域。

邻接表与邻接矩阵的关系如下：

(1) 对应于邻接矩阵的每一行有一个线形链接表；

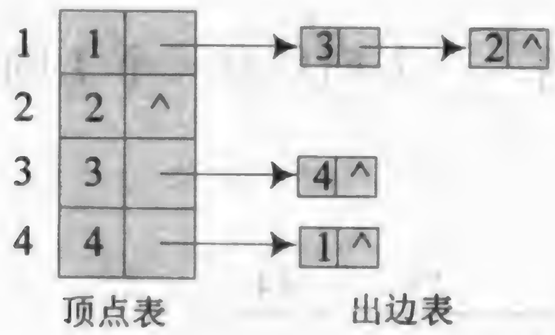
(2) 链接表的表头对应着邻接矩阵该行的顶点；

(3) 链接表中的每个结点对应着邻接矩阵中该行的一个非零元素；

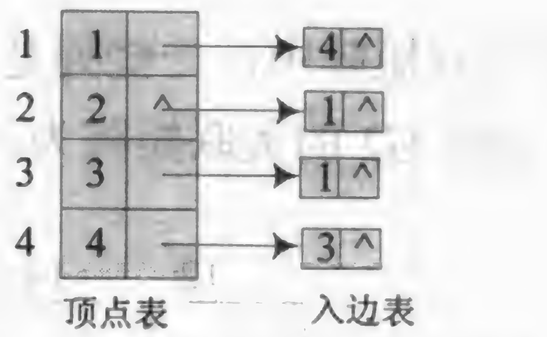
(4) 对于无向图，一个非零元素表示与该行顶点相邻接的另一个顶点；

(5) 对于有向图，非零元素则表示以该行顶点为起点的一条边的终点。

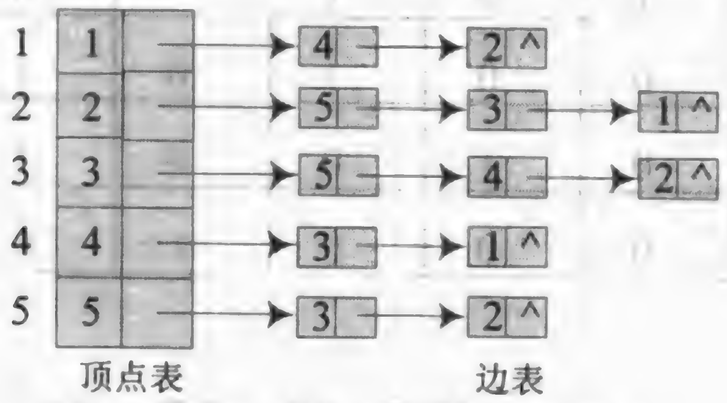
邻接表表示法示例如图 6-7 所示。



(a) 图 6-1(a)有向图 *G*1 的邻接表



(b) 图 6-1(a)有向图 *G*1 的逆邻接表



(c) 图 6-1(b)有向图 *G*2 的邻接表

图 6-7 有向图和无向图的邻接表

邻接表的性质如下：

(1) 图的邻接表表示不是惟一的，它与表结点的链入次序有关；

(2) 无向图的邻接表中第 i 个边表的结点个数即为第 i 个顶点的度；

(3) 有向图的邻接表中第 i 个出边表的结点个数即为第 i 个结点的出度，有向图的逆邻接表中第 i 个入边表的结点个数即为第 i 个结点的入度；

(4) 无向图的边数等于邻接表中边表结点数的一半，有向图的弧数等于邻接表(逆邻接表)中出边表结点(入边表结点)的数目。

需要说明的是：

(1) 在邻接表的每个线性链接表中各结点的顺序是任意的；

(2) 邻接表中的各个线性链接表不说明它们顶点之间的邻接关系；

(3) 对于无向图，某顶点的度数=该顶点对应的线性链表的结点数；

(4) 对于有向图，某顶点的“出度”数=该顶点对应的线性链表的结点数；求某顶点的“入度”需要对整个邻接表的各链接扫描一遍，看有多少与该顶点相同的结点，相同结点数之和即为“入度”值。

6.2.3　关联矩阵

图的另一种矩阵表示法为以顶点和边的关联关系为基础建立矩阵，这个矩阵称之为关联矩阵。定义如下：

图 $G=(V,E)$的关联矩阵是一个$|V|\times|E|$矩阵，使得：

$$a_{ij}=\begin{cases}1 & \text{边}e_i\text{和顶点}v_i\text{关联1次}\\ 0 & \text{边}e_i\text{和顶点}v_i\text{不关联}\\ 2 & \text{边}e_i\text{和顶点}v_i\text{关联2次}\end{cases}$$

在一个多图的关联矩阵中，一些列是相同的，一个列只有一个 1 则代表一个环。图 6-1(b)中无向图 *G*2 的关联矩阵为：

	12	14	23	25	34	35
1	1	1	0	0	0	0
2	1	0	1	1	0	0
3	0	0	1	0	1	1
4	0	1	0	0	1	0
5	0	0	0	1	0	1

6.3　图 的 遍 历

本节讲述的主要内容为图的深度优先搜索遍历(DFS)和广度优先搜索遍历(BFS)。

从图中某个顶点出发访问图中所有顶点，且使得每一顶点仅被访问一次，这一过程称之为图的遍历。图的遍历是图的运算中最重要的运算，图的许多运算均以遍历为基础。

图的遍历按搜索路径不同分为深度优先搜索遍历(Depth First Search)和广度优先搜索遍历(Breadth First Search)。

对每种搜索顺序，访问各顶点的先后次序也不是惟一的。为了避免同一顶点被多次访问，在图的遍历过程中必须记住每个被访问过的顶点，一般可设一数组(如 visited)为标志，以标识顶点是否被访问过。若访问过某顶点则相应的数组元素为真，否则为假。

6.3.1　深度优先搜索遍历

假定给定图 *G* 的初态是所有顶点均未曾访问过，在 *G* 中任选一顶点 *v* 为初始出发点，则深度优先搜索可定义如下：从指定的起点 *v* 出发(先访问 *v*，并将其标记为已访问过)，访

问它的任意相邻接的顶点 w_1，再访问 w_1 的任一个未访问的相邻接顶点 w_2，如此下去，直到某顶点已无被访问过的邻接顶点或者它的所有邻接顶点都已经被访问过了，就回溯到它的前趋。如果这个访问和回溯过程返回到遍历开始的顶点，就结束遍历过程。如果图中仍存在一些未访问过的结点，就另选一个未访问过的结点重新开始深度优先搜索遍历。

可见，图的深度优先搜索遍历是一个递归过程，其特点是尽可能先对纵深方向的顶点进行访问。

对图进行深度优先搜索遍历时，按访问顶点的先后次序所得到的顶点序列称为该图的深度优先搜索遍历序列，简称为 DFS 序列。一个图的 DFS 序列不一定惟一，这与算法、图的存储结构以及初始出发点有关。

深度优先搜索遍历算法表示如下：

```
DFS(v)
    num(v)=i++;
    for 所有与 v 邻接的顶点 u
        if num(u)是 0
            将 edge(uv)连接到 edges 中;
    DFS(u);

depthFirstSearch()
    for 所有向量 v
        num(v)=0;
    edges=null;
    i=1;
    while 有一个向量 v 使得 num(v)是 0
        DFS(v);
    输出 edges;
```

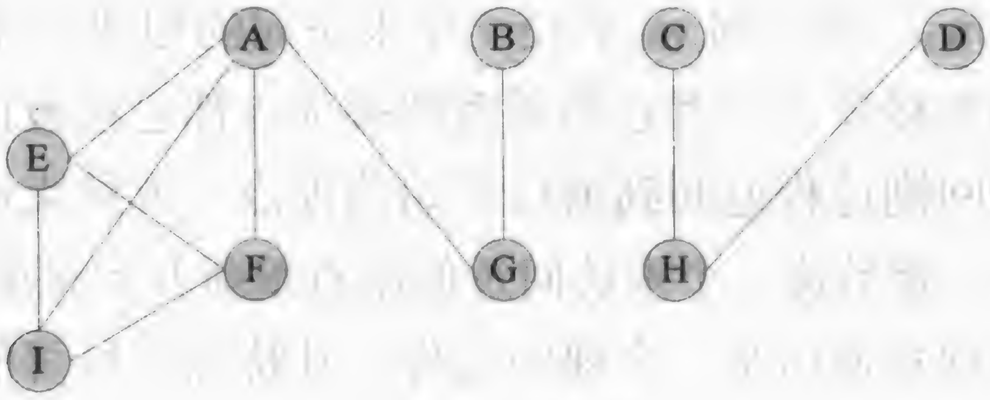
图 6-8 包含了一个实例，利用上述算法，为每个顶点赋了一个数值 num(v)，标在括号内。在完成所有必须的初始化后，depthFirstSearch()调用 DFS(a)。

(1) 第一次用顶点 *a* 调用 DFS()，num(a)赋值为 1。*a* 有 4 个邻接顶点，选择顶点 *e* 进行下一次调用 DFS(e)，为这个顶点赋值 2，也就是 num(e)=2，并将 edge(ae)放入 edges。

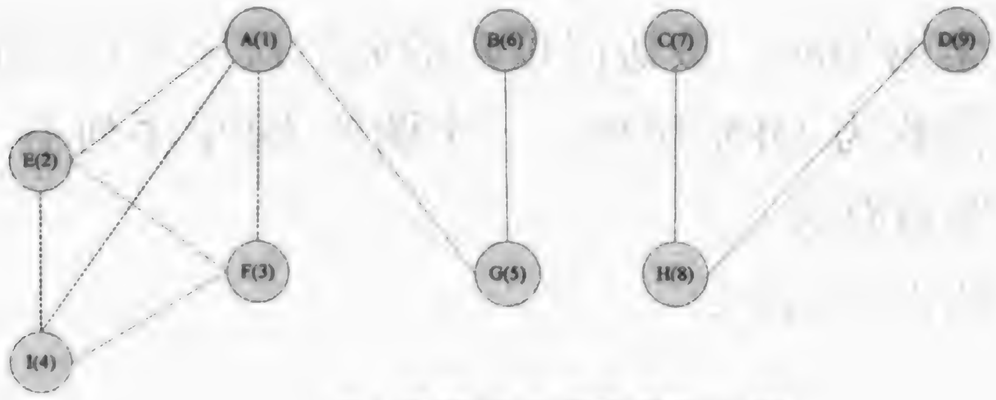
(2) 顶点 *e* 有两个未访问的相邻顶点，先用第一个顶点 *f* 调用 DFS()。DFS(f)调用产生了赋值语句 num(f)=3 并将 edge(ef)放入 edges。

(3) 顶点 *f* 只有一个未访问的相邻顶点 *i*，因此，第 4 个调用 DFS(i)产生赋值语句 num(i)=4 并将 edge(fi)加进 edges。顶点 i 的相邻顶点都已经访问过了，因此，返回调用 DFS(f)后又返回到 DFS(e)，这里只要知道 num(i)不为 0 就知道顶点 *i* 已经访问过了，这也是不将边(ei)添加进 edges 中的原因。

余下的执行步骤可以参阅图 6-8(b)。实线标志着集合 edges 中的边。



(a) 一个无向图

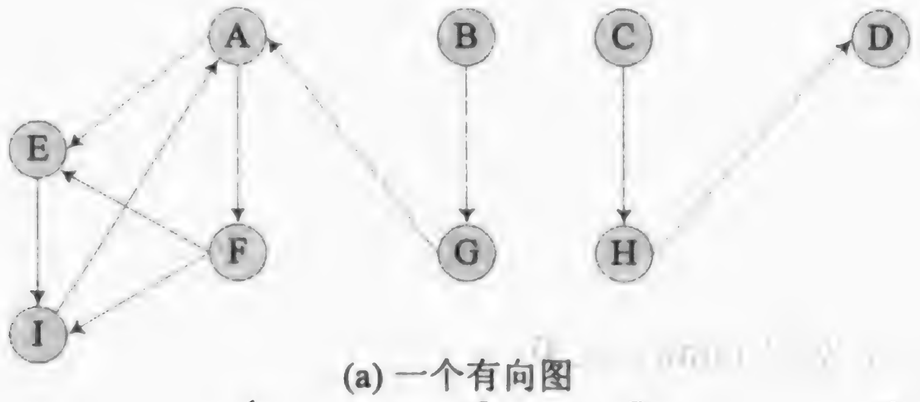


(b) 深度优先搜索遍历示例

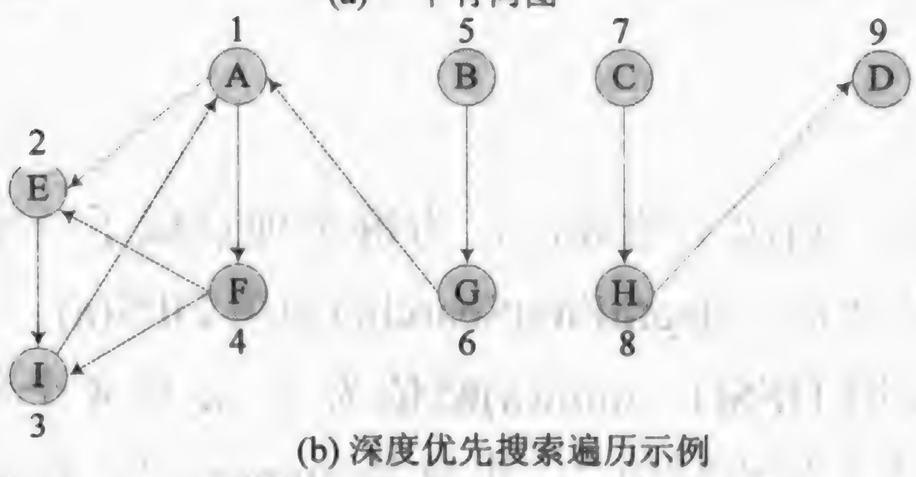
图 6-8　无向图中应用 depthFirstSearch()算法的示例

这个算法保证生成一个树(或是一个森林，森林是树的集合)，它包含或覆盖了原图的所有顶点。

图 6-9 说明了上述算法在有向图中的执行。



(a) 一个有向图



(b) 深度优先搜索遍历示例

图 6-9　有向图中应用 depthFirstSearch()算法的示例

depthFirstSearch()的复杂性是$O(|V|+|E|)$，因为：

(1) 为每个顶点 v 初始化 num(v)需要$|v|$步。

(2) 为每个顶点 v 调用 DFS(v)共 n 次，其中 n 是 v 的边数；每条边调用一次，则产生更多的调用或者结束递归调用链。因此，总的调用次数是 $2|E|$。

(3) 语句中需要查找顶点。

while 有一个向量 v 使得 num(v)是 0。

假定它需要$|v|$步。对于一个没有孤立顶点的图，循环只需要重复一次，每步都能找到一个初始状态的顶点，尽管查找可能需要$|v|$步。对于一个所有的顶点都孤立的图，循环重复$|v|$次，每次中每步也能选择一个顶点，虽然这种实现不能令人满意，但第 i 次重复可能需要 i 步，所以循环总共需要$O(|V|^2)$步。

6.3.2 广度优先搜索遍历

设图 G 的初态是所有顶点均未访问过，在 G 中任选一顶点 v 为初始出发点，则广度优先搜索遍历的基本思想是：从指定的起点 v 出发，访问与它相邻的所有顶点 w_1，w_2，……；然后再依次访问 w_1，w_2，……邻接的尚未被访问的所有顶点，再从这些顶点出发访问与它们相邻接的尚未被访问的顶点，直到所有顶点均被访问过为止。如果图中仍存在一些未访问过的结点，就另选一个未访问过的结点重新开始广度优先搜索遍历。

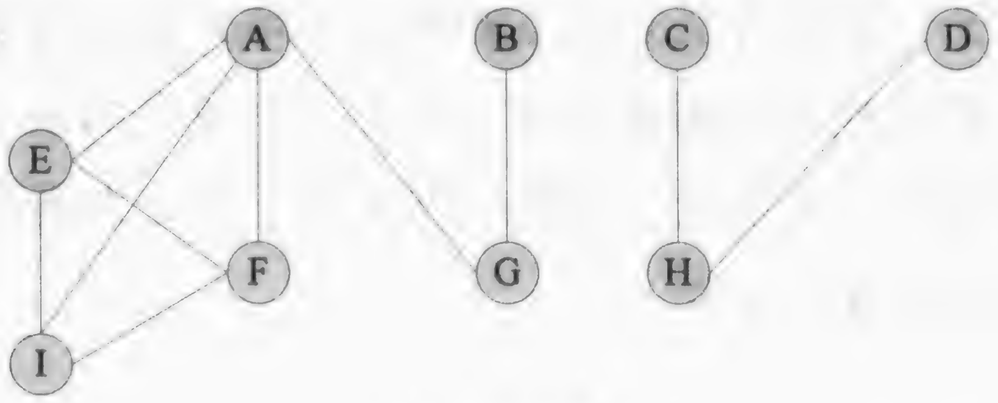
可见，图的广度优先搜索遍历是一个递归过程，其特点是尽可能先对横向的顶点进行访问。

对图进行广度优先搜索遍历时，按访问顶点的先后次序所得到的顶点序列，称为该图的广度优先搜索遍历序列，简称为 BFS 序列。一个图的 BFS 序列不一定惟一，这与算法、图的存储结构以及初始出发点有关。

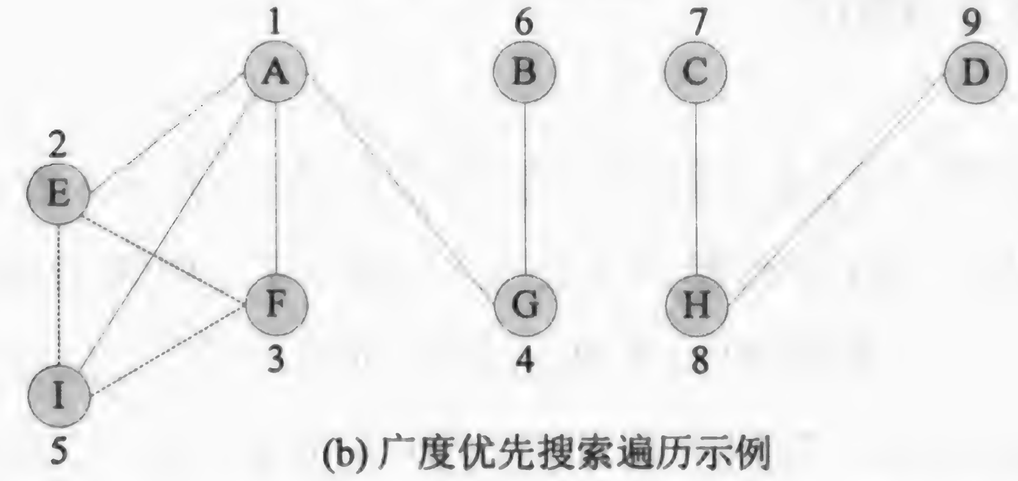
广度优先搜索遍历算法(以队列作为基本数据结构)表示如下：

```
breadthFirstSearch()
    for 所有顶点 u
        num(u)=0;
    edges=null;
    i=l;
    while 存在一个顶点 v 使得 num(v)==0
        num(v)=i++;
        enqueue(v); //进入队列
        while 队列非空
            v=dequeue();
    for 所有和 v 邻接的顶点 u
        if num(u)是 0
            num(u)=i++;
        enqueue(u);
        将 edge(vu)连接到 edges 中;
    输出 edges;
```

图 6-10 和图 6-11 分别显示了处理一个简单图和一个有向图的例子。

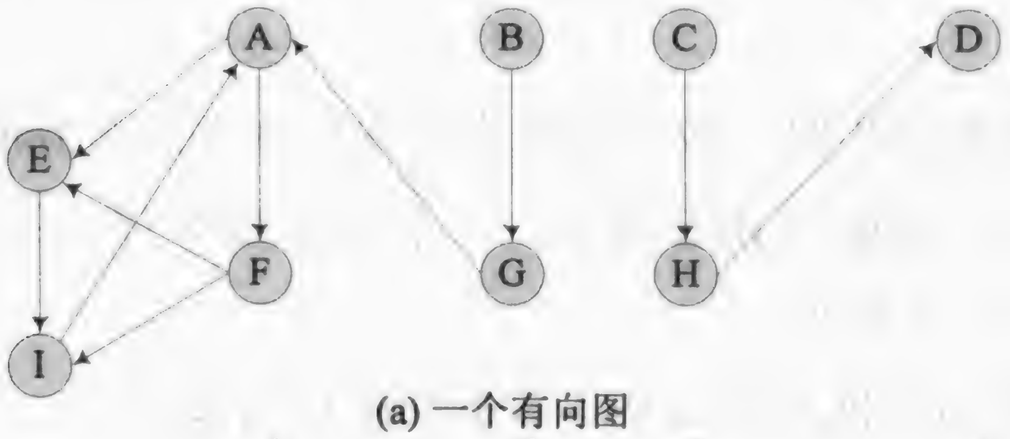


(a) 一个无向图

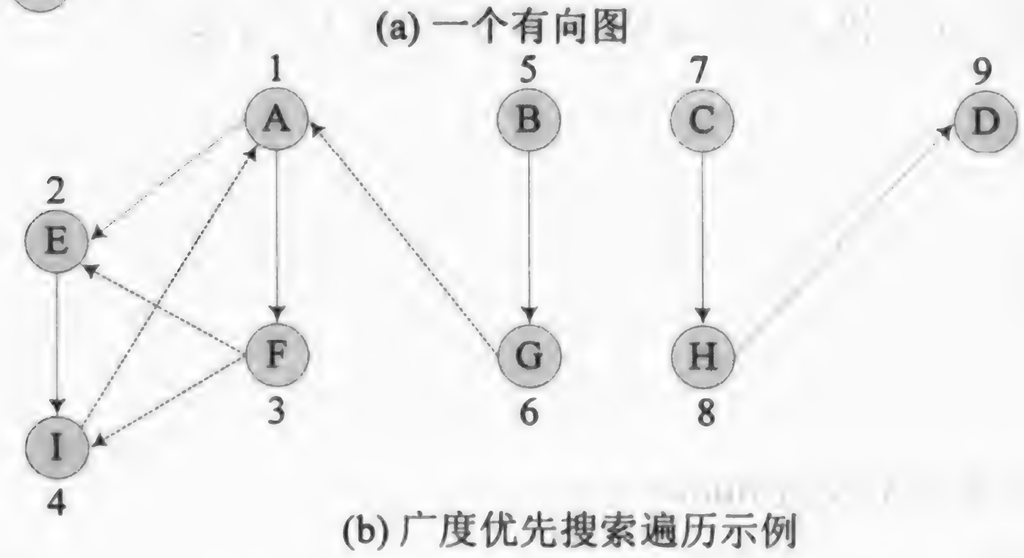


(b) 广度优先搜索遍历示例

图 6-10　无向图中应用 breadthFirstSearch()算法的示例



(a) 一个有向图



(b) 广度优先搜索遍历示例

图 6-11　有向图中应用 breadthFirstSearch()算法的示例

breadthFirstSearch()在处理其他顶点之前先标记顶点 v 的所有相邻顶点，而 DFS()只选择 v 的一个相邻顶点，先不去处理 v 的其他相邻顶点而是去找所选的这个相邻顶点的相邻顶点。

6.4　最小生成树

本节讲述的主要内容为最小生成树以及普里姆(Prim)算法、克鲁斯卡尔(Kruskal)算法。图论中，通常将树定义为一个无回路连通图。对于无回路连通图，只要选定某个顶点

作为根，以此顶点为树根对每条边定向，就能得到通常的树。

6.4.1　生成树

一个连通图 G 的子图如果是一棵包含 G 的所有顶点的树，则该子图称为 G 的生成树。n 个顶点的连通图 G 的任何生成树一定是包含 n 个顶点和 $n-1$ 条边的连通子图(称为 G 的极小连通子图)，反之亦然。

因为树被视作一个无回路的连通图。一个包含 n 个顶点的连通图至少含 $n-1$ 条边(否则不连通)，另一方面要使得图中无回路则至多包含 $n-1$ 条边。

从以上描述中可以看出生成树具有以下特点：

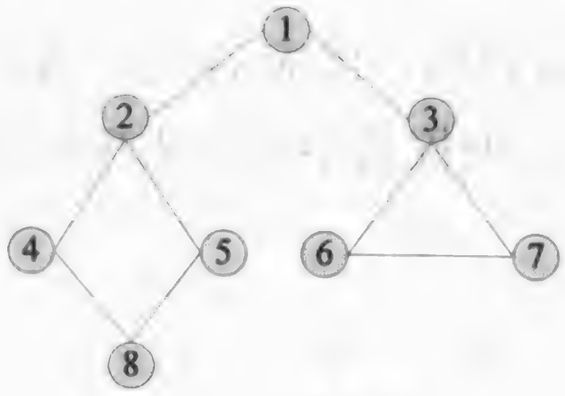
(1) 如果在生成树中去掉任何一条边，此子图就会变成非连通图；

(2) 任意两个顶点之间有且仅有一条路径，如再增加一条边就会出现一条回路；

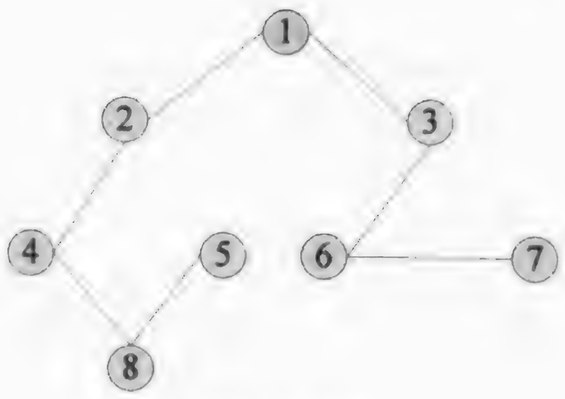
(3) 由遍历连通图 G 时所经过的边和顶点构成的子图是 G 的生成树。

图 G 是一个具有 n 个顶点的连通图，则从 G 的任一顶点出发，作一次深度优先搜索或者广度优先搜索，就可将 G 中的所有 n 个顶点都访问到。显而易见，在这两种搜索算法中，从一个已经访问过的顶点搜索到一个未曾访问过的邻接点，必定要经过 G 中的一条边；而这两种算法对图中的 n 个顶点都仅访问过一次，因此除初始出发点外，对其余 $n-1$ 个顶点的访问一共要经过 G 中的 $n-1$ 条边，这 $n-1$ 条边将 G 中的 n 个顶点连接成 G 的极小连通子图，从而得到 G 的一棵生成树。

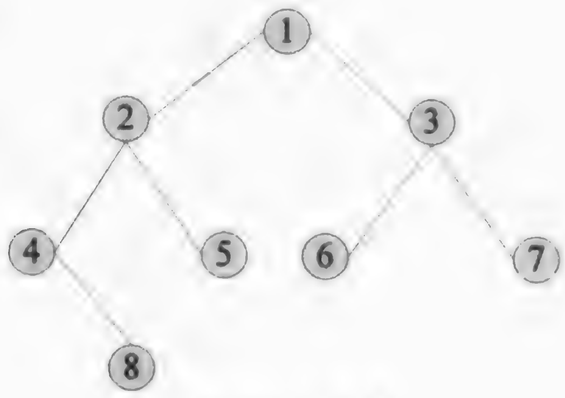
由深度优先搜索得到的生成树称为深度优先生成树，简称为 DFS 生成树；由广度优先搜索得到的生成树称为广度优先生成树，简称为 BFS 生成树。生成树示例如图 6-12 所示。



(a) 一个连通图



(b) DFS 生成树

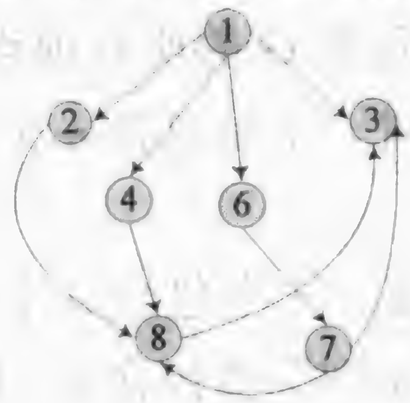


(c) BFS 生成树

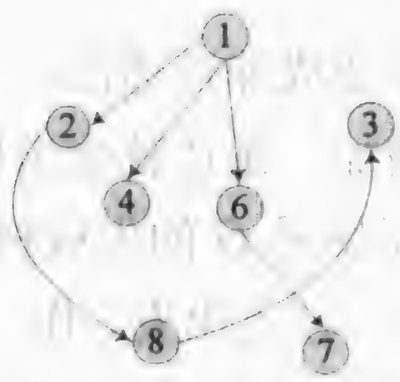
图 6-12　生成树示例

由于从图的遍历可求得生成树，如果将生成树定义为：若从图的某个顶点出发，可以系统地遍历图中的所有顶点，则遍历时经过的边和图的所有顶点所构成的子图，称为图的

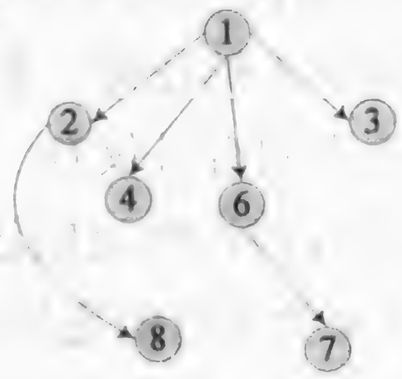
生成树。这样，如果 G 是强连通的有向图，则从其中任何一顶点 v 出发，均可遍历 G 中所有顶点，从而得到一棵以 v 为根的生成树。如果 G 是有根的有向图，设根为顶点 1，则从根 1 出发也可完成对 G 的遍历，从而得到 G 的以顶点 1 为根的生成树。如图 6-13 所示是以顶点 1 为根的有向图及其 DFS 生成树和 BFS 生成树。



(a) 以 1 为根的有向图



(b) DFS 生成树



(c) BFS 生成树

图 6-13　有向图及其生成树

6.4.2　最小生成树的生成

1. 最小生成树

对于连通网络 $G=(V,E)$，边是带权的，因而 G 的生成树的各边也是带权的。生成树的各边的权值总和称为生成树的权，并把权最小的生成树称为 G 的最小生成树。图 6-14 所示为一个连通网络。

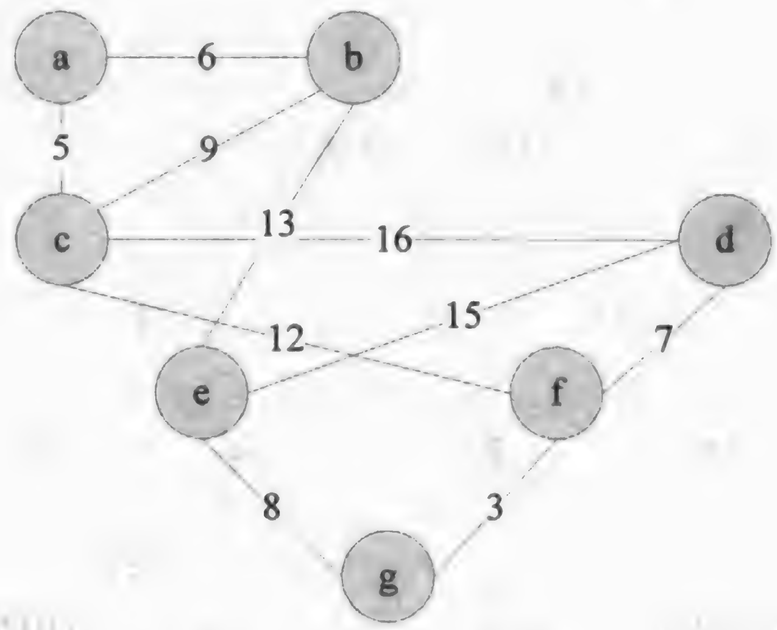


图 6-14　连通网络

构成最小生成树的方法有多种。这些算法可以分成下面几类：

(1) 创建并扩展一些树，使它们合并成更大的树；

(2) 扩展一个树的集构成一棵生成树，如 Kruskal 算法；

(3) 创建并扩展一棵树，为它添加新的树枝，如 Prim 算法；

(4) 创建并扩展一棵树，为它添加新的树枝，也可能从中删除一些树枝。

无论哪一类型的算法均用到了最小生成树如下所述的性质。

2. MST 性质

设 $G=(V,E)$是一个连通网络，U 是顶点集 V 的一个真子集。如果(u,v)是 G 中所有的一个端点在 U(即 $u\in U$)里，另一个端点不在 U(即 $v\in V-U$)里的边中，具有最小权值的一条边，则一定存在 G 的一棵最小生成树包括此边(u,v)。这个性质称之为 MST 性质。

MST 性质用反证法证明如下：

假设 G 的任何一棵最小生成树中都不包含边(u,v)。设 T 是 G 的一棵最小生成树，T 不包含边(u,v)。由于 T 是树，是连通的，因此有一条从 u 到 v 的路径；而且该路径上必有一条连接两个顶点集 U 和 $V-U$ 的边(u',v')，其中 $u'\in U$，$v'\in V-U$，否则 u 和 v 不连通。当把边(u,v)加入树 T 时，得到一个包含有边(u,v)的回路，如图 6-15 所示。删除边(u',v')，上述回路即被删除，由此得到另一棵生成树 T'，T'和 T 区别仅在于用边(u,v)取代了 T 中的边(u',v')。因为(u,v)的权小于或者等于(u',v')的权，所以 T'的权小于或者等于 T 的权，因此 T'也是 G 的最小生成树，它包含边(u,v)，与假设矛盾。

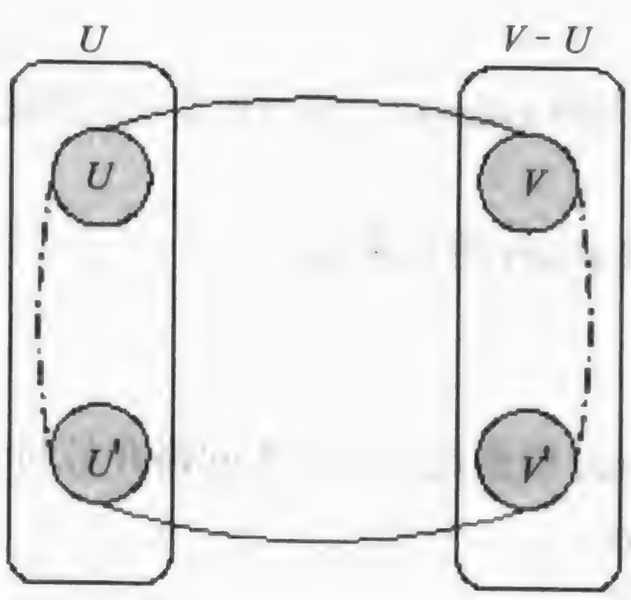


图 6-15　包含(u,v)的回路

3. 普里姆(Prim)算法

设 $G(V,E)$为一个连通网，顶点集 $V=(v_1,v_2,\cdots\cdots,v_n)$。设 $T(U,TE)$是所要求的 G 的一棵最小生成树，其中 U 是 T 的顶点集，TE 是 T 的边集，并且将 G 中边上的权看作是长度。

普里姆算法的基本思想：首先任选 V 中一顶点(不妨为 v_1)，构成入选顶点集 $U=\{v_1\}$，此时入选边集 TE 为空集，V 中剩余顶点构成待选顶点集 V-U；在所有关联于入选顶点集和待选顶点集的边中选取权值最小的一条边(v_i,v_j)加入入选边集(这里 v_i 为入选顶点，v_j 为待选顶点)，同时将 v_j 加入入选顶点集。重复以上过程，直至入选顶点集 U 包含所有顶点($U=V$)，入选边集包含 $n-1$ 条边，MST 性质保证上述过程求得的 $T(U,TE)$是 G 的一棵最小生成树。

显然普里姆算法的关键是如何找到连接 U 和 $V-U$ 的最短边来扩充生成树 T。一个简

单的方法就是在实施算法之前，将所有的边进行排序。准备加入生成树的边不仅不会在树中产生环路，而且也和树中已有的一个顶点关联。如图 6-16 表示用普里姆算法找到的一棵最小生成树的过程。

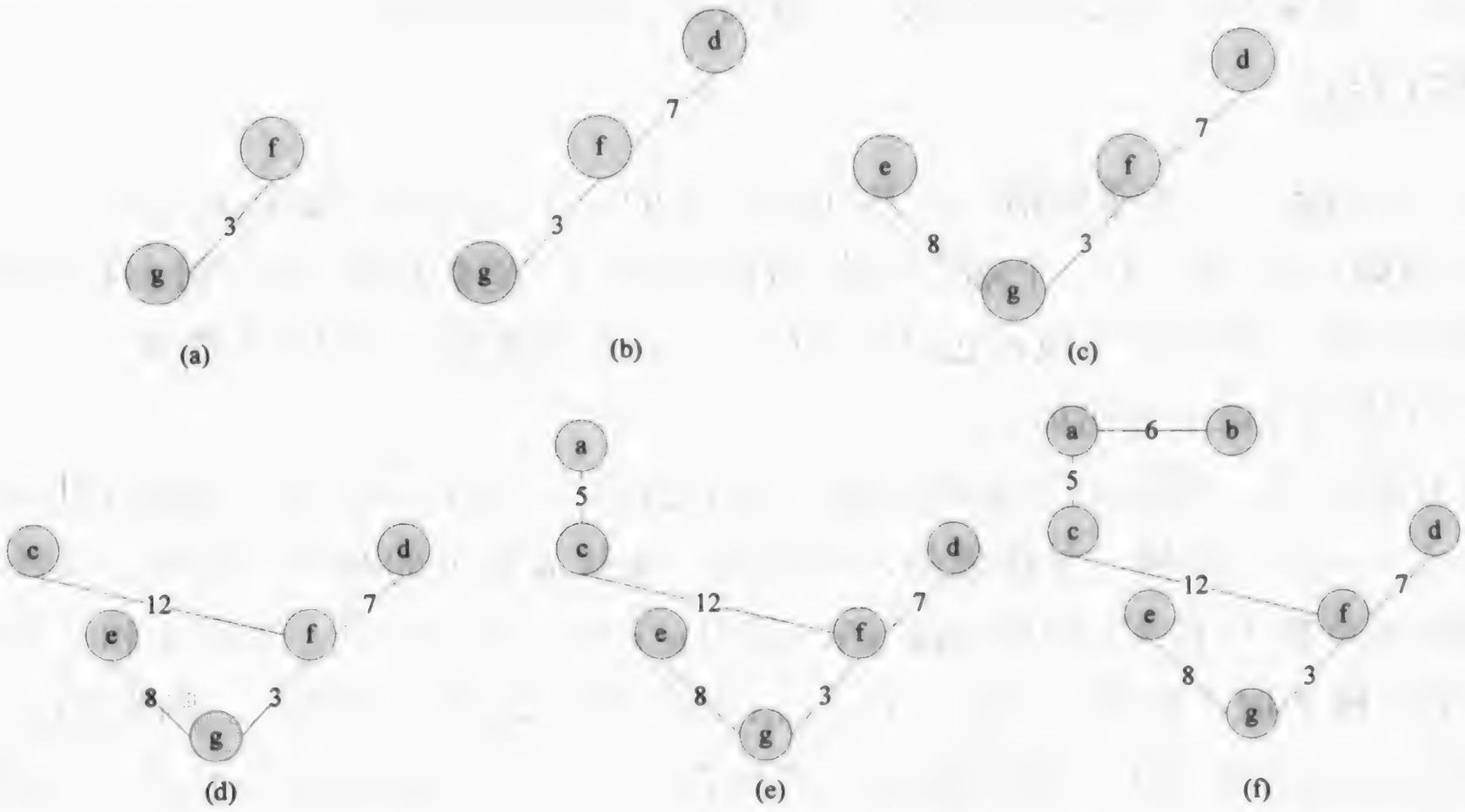


图 6-16　图 6-14 中连通网络利用 Prim 算法找到的一棵生成树

普里姆算法伪代码描述如下：

```
PrimAlgorithm(带权连通无向图 graph)//开始时所有的边都是排序的
tree=null;
edges=按照权值大小排序的 graph 的全部边;
for i=1 到|V|-1
    for j=1 到|edges|
        if edge 中的边 ej 和 tree 中的边不会产生回路并且和 tree 中的一个顶点关联
            将 ej 添加进 tree;
            break;
```

4. 克鲁斯卡尔(Kruskal)算法

设 $G=(V,E)$是连通网络，令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T=(V,\Phi)$，T 中的每个顶点自成一个连通分量。按照长度递增的顺序依次选择 E 中的边(u,v)，如果该边端点 u、v 分别是当前 T 的两个连通分量 T_1、T_2 中的顶点，则将该边加入到 T 中，T_1、T_2 也由此边连接成一个连通分量；如果 u、v 是当前同一个连通分量中的顶点，则舍去此边，这是因为每个连通分量都是一棵树，此边添加到树中将形成回路。依此类推，直到 T 中所有顶点都在同一连通分量上为止。从而得到 G 的一棵最小生成树 T。

同样，在这个算法中，所有的边都是根据权排序的，然后检测这个排序序列中的每条边，如果在构造时，加入它不会产生环路就将它添加到树中。

克鲁斯卡尔算法伪代码描述如下：

KruskalAlgorithm(带权连通无向图 graph)//开始时所有的边都是排序的
　　tree=null;
　　edges=按照权值大小排序的 graph 的全部边;
　　for (i=1;i≤$|E|$ and $|tree|<|V|+1$;i++)
　　　　if edges 中的边 e_i 和 tree 中的边不会产生环路
　　　　　　将 e_i 添加进 tree;

如图 6-17 表示用克鲁斯卡尔算法找到的一棵最小生成树的过程。

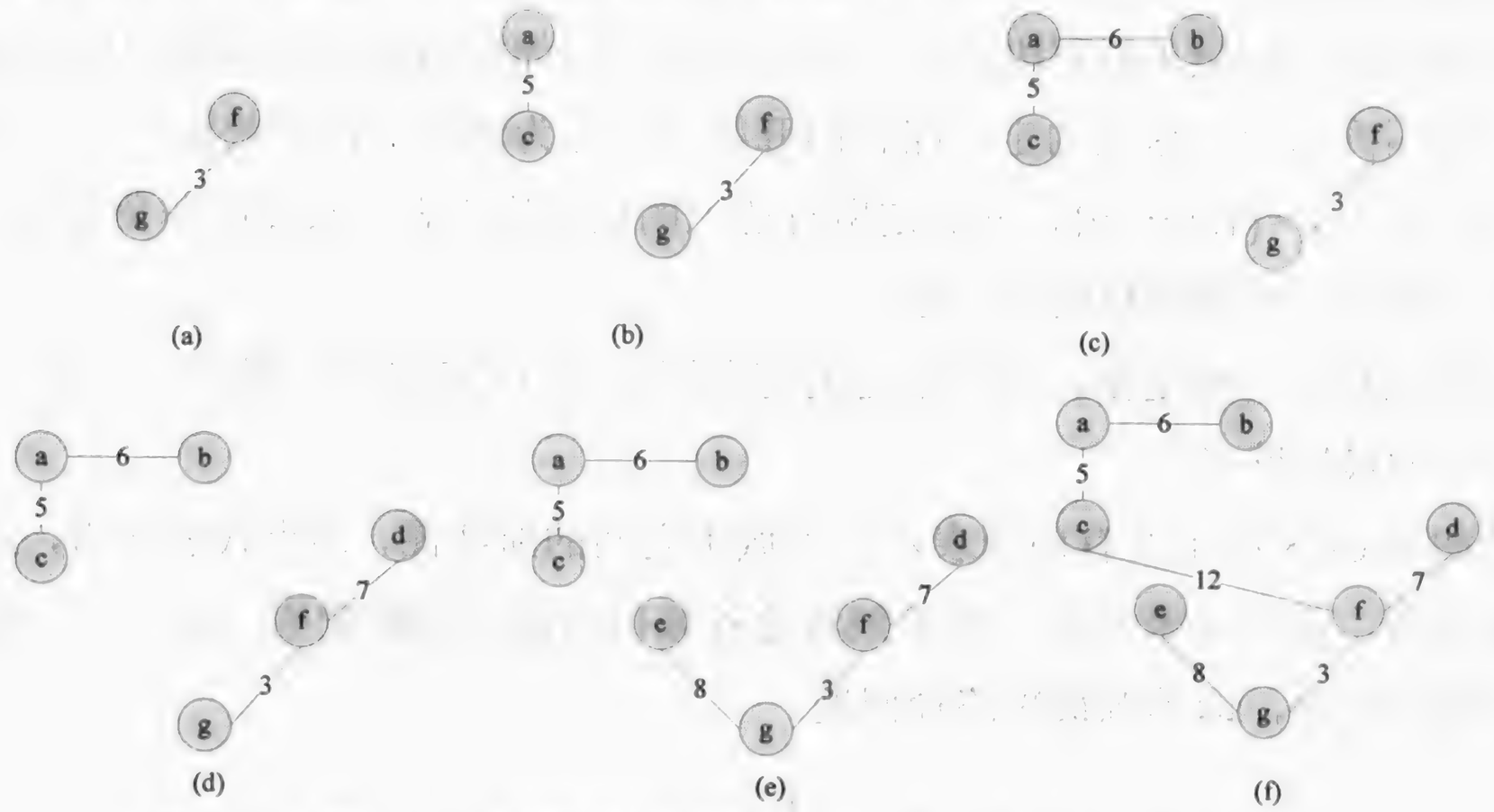


图 6-17　图 6-14 中连通网络利用 Kruskal 算法找到的一棵生成树

克鲁斯卡尔算法和普里姆算法产生的生成树是相同的，不同之处在于边加入树顺序不同，而且普里姆算法总是保持构造中的树是一片的，因此在普里姆算法应用的整个阶段中它都是一棵树。克鲁斯卡尔算法在执行过程中不能保持是一棵树，可能至多是树的集合，但是每条边在克鲁斯卡尔算法中只需要考虑一次，因为如果它在一步当中产生环路，则在以后的步骤中更会产生环路，因此就不用重复考虑了。从这里可以看出克鲁斯卡尔算法更快。

6.5　最短路径和拓扑排序

本节讲述的主要内容为最短路径和拓扑排序。

最小生成树是无向网的一个典型应用。本节重点介绍最短路径和拓扑排序，它们是有向网和有向图的应用。

6.5.1　最短路径

带权图中求最短路径问题，即求两个顶点间长度最短的路径，对现实生活中，如交通网络问题的解决具有重要的意义。这里的路径长度不是指路径上边数的总和，而是指路径上各边的权值总和，这里的权值可以代表距离、运费等具有实际含义的有效数值。

设 A 城到 B 城有一条公路，两座城市的海拔不同，A 城高于 B 城。这样如果考虑到上、下坡的车速问题，则边(A,B)和边(B,A)上表示行驶时间的权值也不同，因此边(A,B)和边(B,A)应该是两条不同的边。这里约定：路径的开始顶点称为源点，路径的最后一个顶点称为终点。设顶点集为 $V=\{1,2,\cdots,n\}$，并假定所有边上的权值均为表示长度的非负实数。

这里主要讨论单源最短路径问题。

单源路径最短问题是指：对于给定的有向网络 $G=(V,E)$及单个源点 v，求从 v 到 G 的其余各顶点的最短路径。

假设如图 6-18 所示的有向网表示 5 个城市之间的航线图，顶点代表城市，弧上的权值代表运输费用，现在要求从某一城市到其他各城市的最小运输费用。这实际上就是求有向网的最短路径问题，即单源最短路径问题。

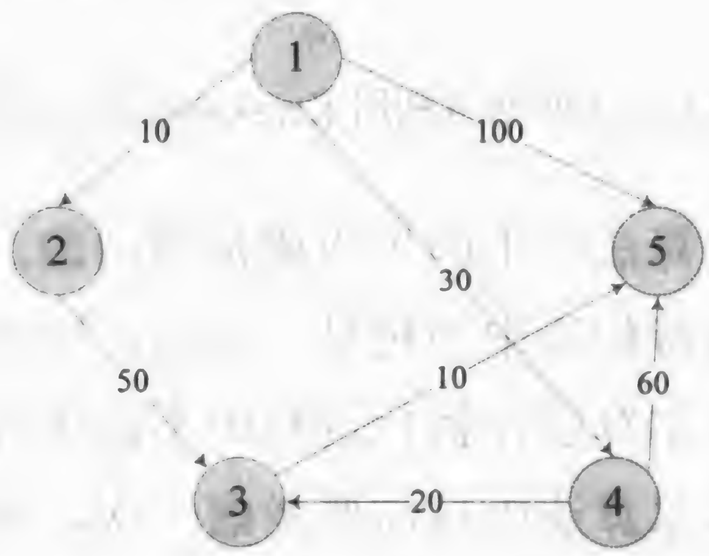


图 6-18　有向网络

在图 6-18 中，设顶点 1 为源点，1 到 2 的路径只有一条：1→2(10)，括号中给出的是该路径上的权值之和，称作路径长度；

1 到 3 的路径有两条：1→2→3(60)，1→4→3(50)；

1 到 4 的路径有一条：1→4(30)；

1 到 5 的路径有四条：1→5(100)，1→4→5(90)，1→2→3→5(70)，1→4→3→5(60)。

选出 1 到其余各顶点的最短路径，并按路径长度递增顺序排列如下：1→2(10)，1→4(30)，1→4→3(50)，　1→4→3→5(60)。

由此可以发现一个规律：按路径长度递增顺序生成从源点到其余各顶点的最短路径时，当前正生成的最短路径上除终点以外，其余顶点的最短路径均已生成。

迪卡斯特拉(Dijkstra)算法正是在上述规律基础上得到的。其基本思想是：设置两个顶

点集 S 和 T，S 中存放已确定最短路径的顶点，T 中存放待确定最短路径的顶点。初始时，S 中仅有一个源点，T 中包含除源点外其余顶点，此时各顶点的当前最短路径长度为源点到该顶点的弧上的权值。接着选取 T 中当前最短路径长度最小的一个顶点 v 加入 S，然后修改 T 中剩余顶点的当前最短路径长度。修改的原则是：当 v 的最短路径长度与 v 到 T 中的顶点之间的权值之和小于该顶点的当前最短路径长度时，用前者替换后者。重复上述过程，直至 S 中包含所有的顶点。

迪卡斯特拉算法伪代码描述如下：

```
S={v};
置 T 中各顶点的距离值;
while S 中顶点数<n
  {
    在 T 中选择距离值最小的顶点 u;
    S=S+{u};
    调整 T 中剩余顶点的距离值;
  }
```

如图 6-19 给出了图 6-18 中有向网络从顶点 1 到其他各顶点最短路径的过程，其中用实线圈表示已确定最短路径的顶点，实线箭头表示已确定距离的最短路径上的弧，顶点旁括号中的数字表示该顶点的当前最短路径长度。

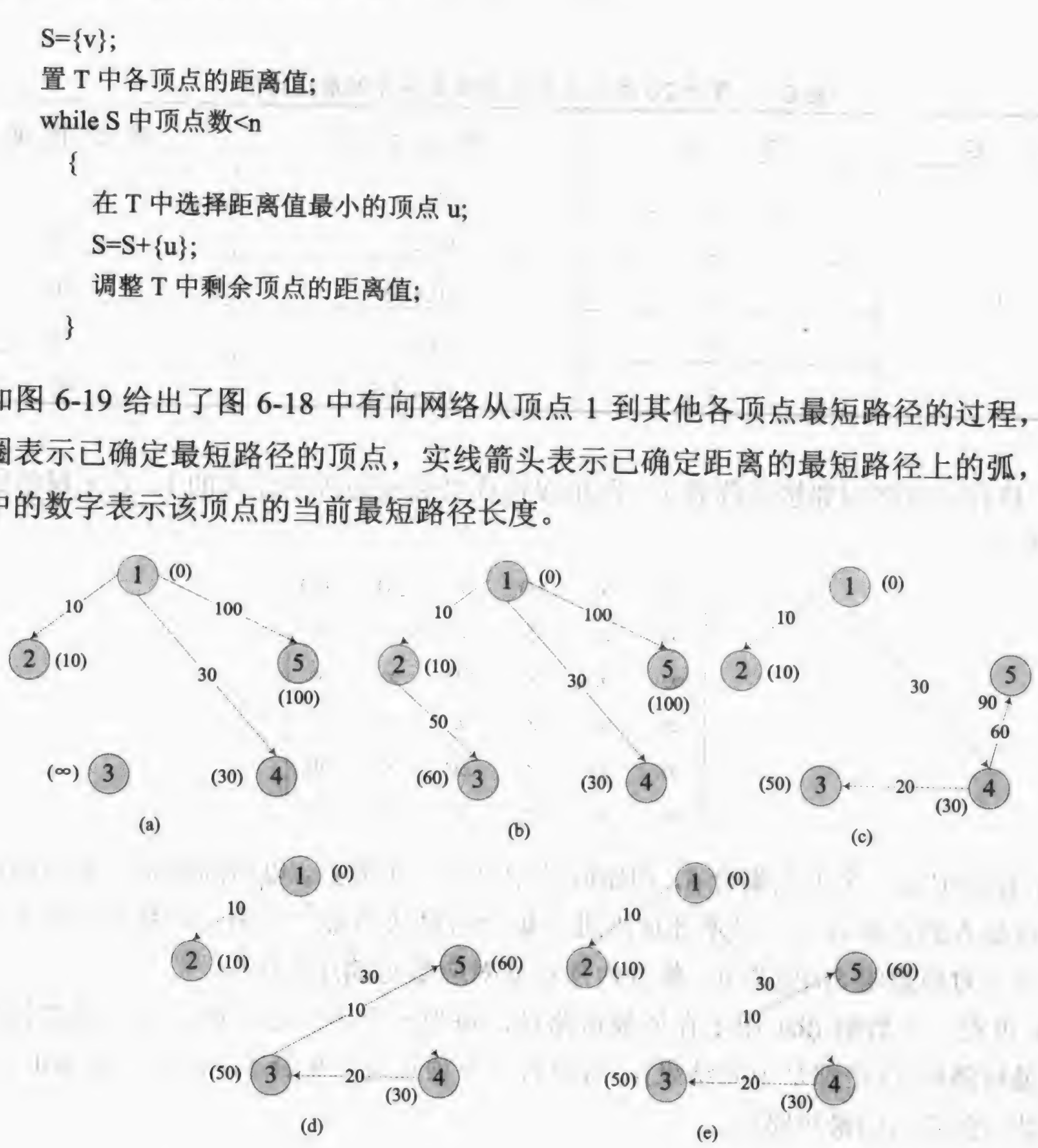


图 6-19　采用迪卡斯特拉算法求图 6-1 中有向图最短路径的过程

下面举例说明迪卡斯特拉算法。

如图 6-20 所示，求从顶点 0 到其余各顶点的最短路径，如表 6-1 所示。

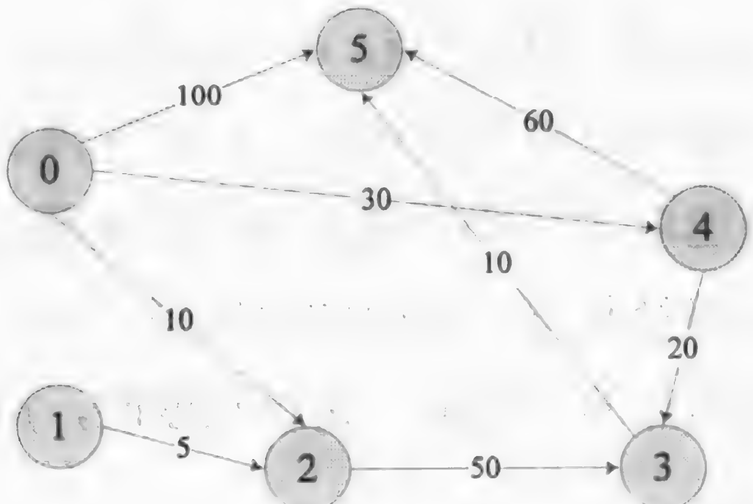


图 6-20　有向网络

表 6-1　图 6-20 中从顶点 0 到其余各点的最短路径

始　点	终　点	最 短 路 径	路 径 长 度
0	1	无	
	2	(0,2)	10
	3	(0,4,3)	50
	4	(0,4)	30
	5	(0,4,3,5)	60

(1) 将有向网络用邻接矩阵表示，即用权值代替邻接矩阵中原来的 1，若无权值的边可用∞来表示。

$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(2) 算法中需一个顶点集合 S，初始时其中只有一个源点，以后陆续将已求得的最短路径的顶点加入到该集合中，当全部顶点进入集合后算法结束。可用一维数组代替集合，集合外顶点 v_i 对应数组 $S[i]$值为 0，集合内顶点 v_i 对应数组 $S[i]$值为 1。

(3) 再设一个数组 dist 用于存放最短路径，每当一个顶点进入集合 S 时就要修改此数组中的最短路径(这些都是中间结果)，当最后一个顶点进入集合 S，再修改完 dist 中的值，即得到某点到各点的最短路径。

迪卡斯特拉算法过程如下：

(1)

	0	1	2	3	4	5
S	1	0	0	0	0	0
dist	0	∞	10	∞	30	100

(2)

	0	1	2	3	4	5
S	1	0	1	0	0	0
dist	0	∞	10	60	30	100

(3)

	0	1	2	3	4	5
S	1	0	1	0	1	0
dist	0	∞	10	30	30	90

(4)

	0	1	2	3	4	5
S	1	0	1	1	1	0
dist	0	∞	10	30	30	60

(5)

	0	1	2	3	4	5
S	1	0	0	0	0	0
dist	0	∞	10	30	30	60

6.5.2 拓扑排序

在现实世界中，需要执行一系列任务。一些任务关系到先执行哪一个，而另一些任务的执行顺序就无关紧要。如房地产项目，可以用一个有向图来描绘其实施过程。显然这个房地产项目可以由若干个子工程或子系统构成，如果把子工程或子系统称为活动(Activity)，这些活动之间就存在先后次序关系，即某项活动的实施必须以另一项活动的完成为前提。我们可以用一个有向图的顶点代表一项活动，用有向图的弧代表活动之间的先后次序关系，即弧代表先决条件，当一项活动 i 是另一项活动 j 的先决条件时，有向图中存在边$<i,j>$。

用顶点表示活动，用弧表示活动之间的先后次序关系的有向图，称为顶点活动图(Activity On Vertex Network)，简称为 AOV 网。可见 AOV 网的特点是在网中一定不能有有向回路。检测网中是否存在环，则采用拓扑排序的方法。

1. 什么是拓扑排序

对于一个 AOV 网，通常需要把它的所有顶点排成一个满足下述关系的线性序列 v_1，v_2，…，v_n，如果 AOV 网中从顶点 v_i 到顶点 v_j 有一条路径，则在该线性序列中顶点 v_i 必在顶点 v_j 之前。满足这种线性关系的序列称为拓扑序列。

对 AOV 网构造拓扑序列的操作称为拓扑排序。即将 AOV 网中各个顶点排列成一个有序序列，使得所有前趋和后继关系都能得到满足，而那些没有次序关系的顶点，在拓扑排序的序列中可以插到任意位置。拓扑排序是对非线形结构的有向图进行线形化的重要手段。

并非任何 AOV 网的顶点都可以排成拓扑序列。如果网中存在有向回路，则找不到该

网的拓扑序列。一般情况下，AOV 网不应该存在有向回路，因为如果存在回路就意味着某项活动的开工是以自己工作的完成为先决条件的，这种死锁的现象会导致项目不可行。而任何无回路的 AOV 网，其顶点都可以排成一个拓扑序列，并且拓扑序列不一定是惟一的。

2. 拓扑排序的算法

拓扑排序的算法的基本步骤是：

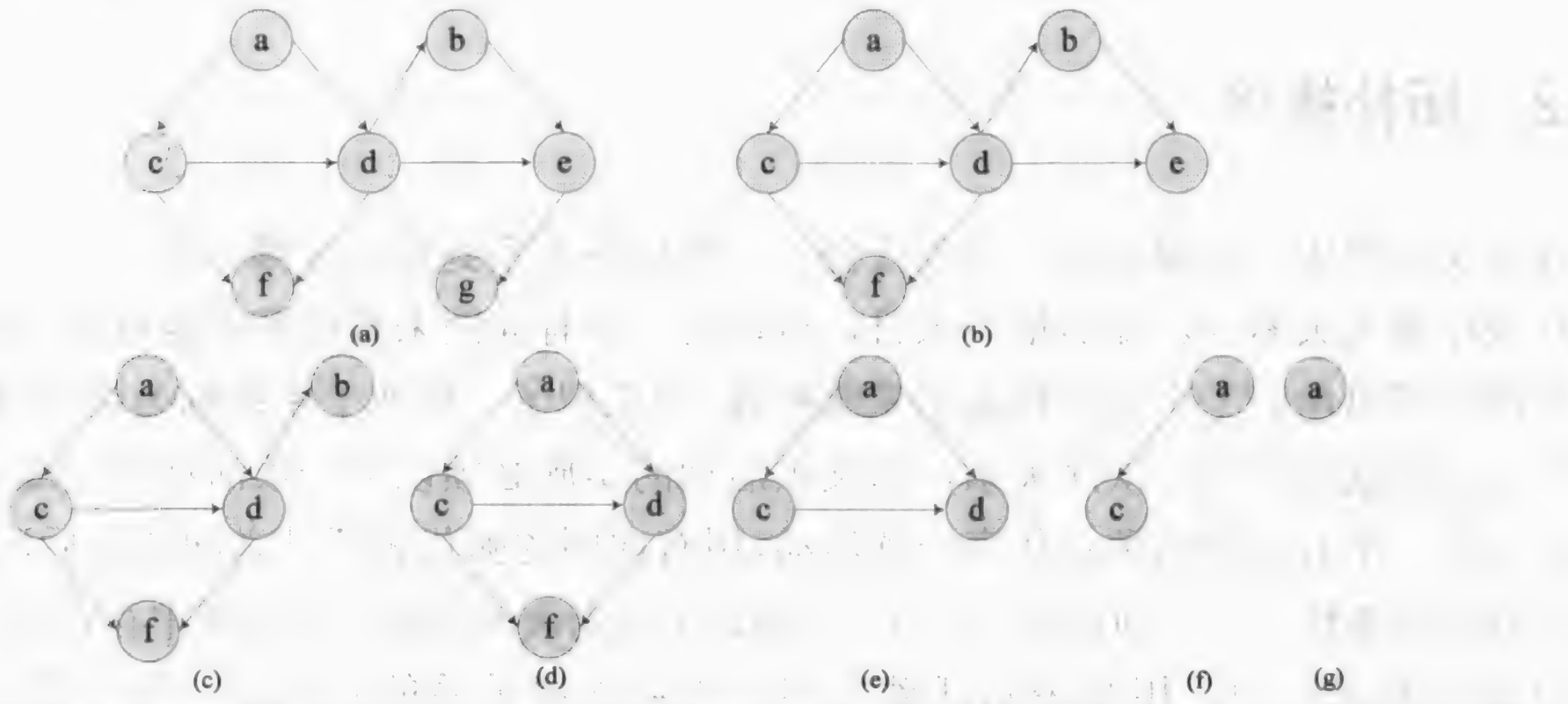
(1) 从网中选择一个入度为 0 的顶点并输出；

(2) 从网中删除此顶点及其所有出边。

其算法描述为：

```
topologicalsort(digraph)
    for i=l 到|V|
        寻找一个最小顶点 v;
        num(v)=I;
        从 digraph 中删除顶点 v 以及与 v 相关联的所有边;
```

如图 6-21 即是这个算法的一个应用示例。图 6-21(a)经过一系列删除产生序列：*g*，*e*，*b*，*f*，*d*，*c*，*a*。



a	1					7
b		4		3		
c		2				6
d		3				5
e			5	2		
f					7	4
g			6	1		

(h)

图 6-21　拓扑排序的执行

对图 6-22(a)中的有向图进行拓扑排序，写出有向图的一个拓扑序列。方法如下：

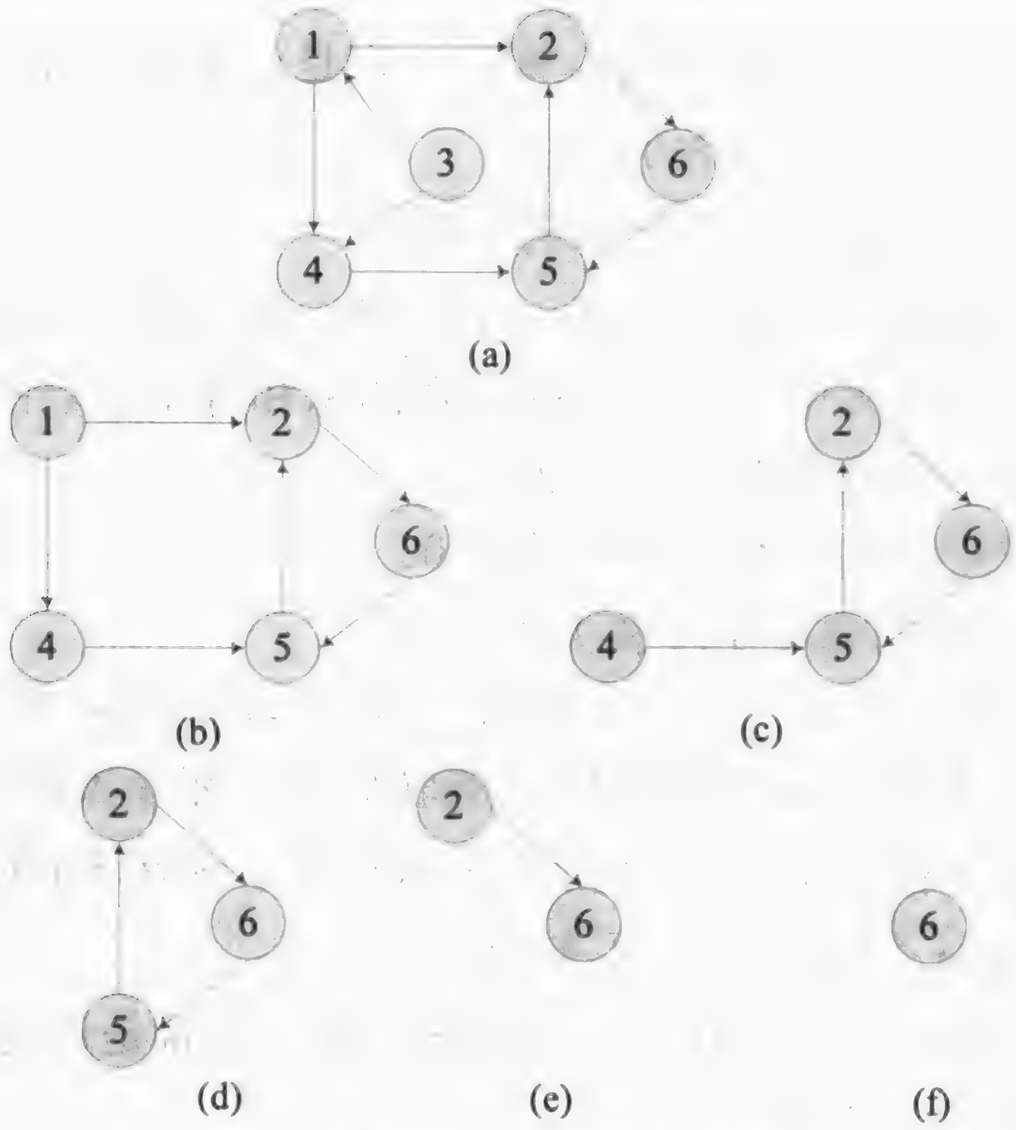


图 6-22 拓扑排序示例

(1) 在图 6-22(a)中的有向图中选取入度为 0 的顶点 3，删除 3 及其相关联的两条弧，如图 6-22(b)所示；

(2) 再在图 6-22(b)中选取入度为 0 的顶点 1，删除 1 及其相关联的两条弧，如图 6-22(c)所示；

(3) 再在图 6-22(c)中选取入度为 0 的顶点 4，删除 4 及其相关联的一条弧，如图 6-22(d)所示；

(4) 再在图 6-22(d)中选取入度为 0 的顶点 5，删除 5 及其相关联的两条弧，如图 6-22(e)所示；

(5) 再在图 6-22(e)中选取入度为 0 的顶点 2，删除 2 及其相关联的一条弧，如图 6-22(f)所示；

(6) 最后选取顶点 6，得到有向图的一个拓扑序列：3，1，4，5，2，6。

思考和练习

(1) 一个图与一个简单图之间有什么区别？

(2) 在一个无向图中，一条边自己可以是一条路径吗？

(3) 为什么一个简单图的定义禁止环，而有向图的定义允许环？

(4) 图 $G=(V,E)$中，边数和所有顶点度数总和的关系是什么？

(5) 画出 n 个点的完全图，n=2，3，4，5，6。

(6) 判断真假：

- 若一个图有 n 个点，$n(n-1)/2$ 条边，则它一定是一个完全图。
- 一条路径的长度一定小于图的大小。
- 一条回路的长度等于它包含的不同点的个数。
- 如果一个图的关联矩阵有 n 条边，$n(n-1)/2$ 列，则该图一定是一个完全图。
- 在一个有向图的关联矩阵中，每一行的项数的总和等于该点的入度。
- 一个图的关联矩阵的所有项的和等于 $2|E|$。
- 一个有向图的关联矩阵的所有项的和等于 0。

(7) 画出 n 个点的完全图的邻接矩阵和关联矩阵。

(8) 一个图(V,E)，若$|E|=\theta(|V|^2)$，称为稠密；若$|E|=o(|V|)$，称为稀疏。

- 对稠密图而言，这 3 种表示(邻接矩阵、关联矩阵、邻接表)哪一种最好？
- 对稀疏图而言，这 3 种表示(邻接矩阵、关联矩阵、邻接表)哪一种最好？

(9) 画出一棵树，它有 n 个顶点，$n-1$ 条边。

(10) 画一个简单图，说明如果它有一个生成树，那么它就是连通的。

(11) 某带权无向图如图 6-23 所示。

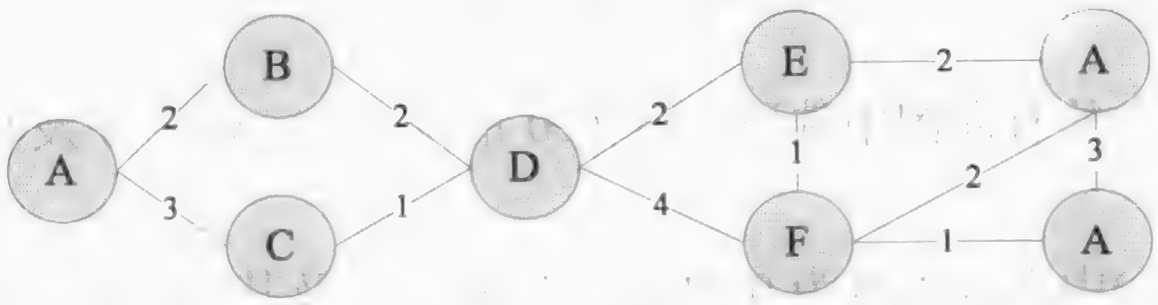


图 6-23　带权无向图

- 给出其邻接矩阵和邻接表表示。
- 画出以 A 为起点的一棵深度优先生成树和广度优先生成树。
- 画出以 D 为起点的一棵深度优先生成树和广度优先生成树。
- 用 Kruskal 算法求其最小生成树。要求画出依次选取每一条边的过程。
- 用 Prim 算法求其最小生成树，并画出每一步骤结果。

(12) Dijkstra 算法是如何应用到无向图中的？

(13) 修改 Dijkstra 算法，使其可以查找从顶点 a 到顶点 b 的最短路径。

(14) 找出图 6-24 顶点 1 到各顶点间的最短路径。

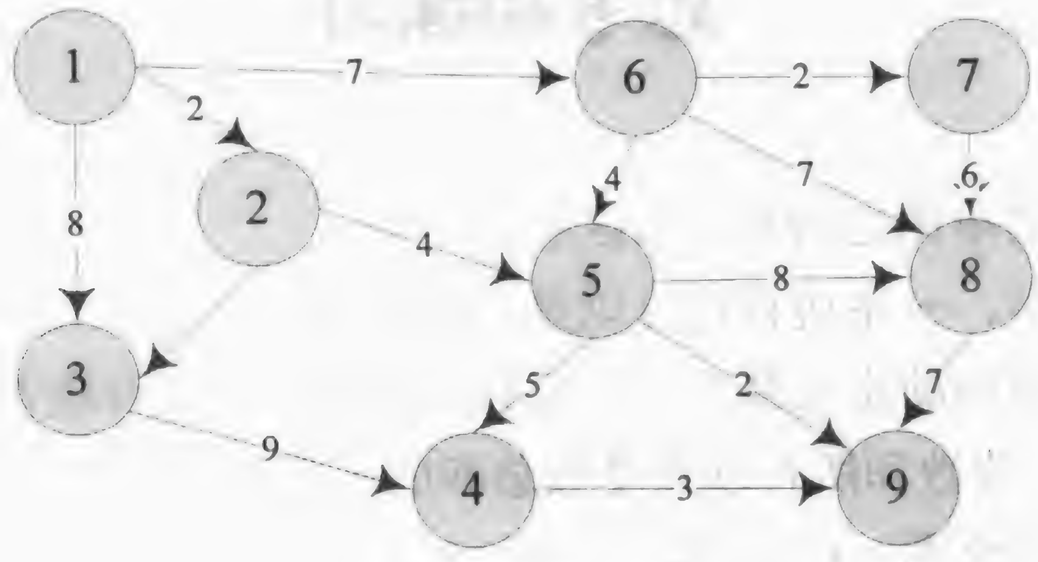


图 6-24　题 14 图

(15) 设有向图 $G=(V,E)$，试设计一个函数 cycle，检测 G 中是否存在回路(环)，若存在回路，则输出回路上的全部顶点。

(16) 如何找到第二小的生成树？

(17) 怎样用最小生成树算法查找最大生成树？

(18) 一个锦标赛是一个有向图，它的每对顶点间恰好有一条边，

- 一个锦标赛有多少条边？
- 可以创建多少个 n 条边的锦标赛？
- 能不能给每个锦标赛进行拓扑排序？
- 一个锦标赛有多少个最小顶点？
- 传递锦标赛是一个锦标赛，如果存在边 edge(vu)和 edge(uw)，则必存在边 edge(vw)。这种锦标赛能不能有环？

(19) 假定 G 是下面邻接矩阵表示的图。

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

- 画出 G。
- G 是简单图吗？
- G 是有向图吗？
- G 是强连通吗？
- G 是弱连通吗？
- G 是无环图吗？

(20) 有 4 种遍历二叉树的算法：前序遍历、中序遍历、后序遍历、同层次遍历。若一个二叉树是一个连通无环图，下面搜索方法属于哪个算法？

- 深度优先搜索
- 宽度优先搜索

第7章　排　　序

排序是计算机程序设计中的一种重要操作，掌握排序的方法，可以在实际的应用过程中提高查找的效率。

本章的学习目标：

- 排序的基本概念；
- 几种内部排序的基本思想、算法和性能；
- 外部排序中的二路归并、置换选择和多路归并算法。

7.1　概　　述

如果数据能够根据某种规则排序，就能大大提高数据处理的算法效率。例如，在一本电话黄页中，如果名称或号码不按一定的规律排序，那么要查找一个名称或号码几乎是不可能的。同样的道理，对于字典、学生名册等其他需要按顺序组织的东西也是一样的。由此可见，使用排序的方便性是毋庸质疑的，而且它对计算机科学也是如此。尽管计算机比人处理无序的数据更容易、更快，但是用它来处理这样的无序数据集是极其低效的。通常在处理数据之前要对其先进行排序。

7.1.1　排序的基本概念

所谓排序就是整理文件中的记录，使之按关键字递增(或递减)的次序排列起来。其确切的定义如下：

假设含 n 个记录的序列为$\{R_1,R_2,\cdots,R_n\}$，其相应的关键字序列为$\{K_1,K_2,\cdots,K_n\}$，需确定 $1,2,\cdots,n$ 的一种排列 $R_{i1},R_{i2},\cdots,R_{in}$，使其相应的关键字满足 $K_{i1}\leqslant K_{i2}\leqslant\cdots\leqslant K_{in}$(或 $K_{i1}\geqslant K_{i2}\geqslant\cdots\geqslant K_{in}$)的关系。

简单地说，使序列$\{R_1,R_2,\cdots,R_n\}$成为一个按关键字有序的序列$\{R_{i1},R_{i2},\cdots,R_{in}\}$，这样的一种操作称为排序。其输入内容为 n 个记录 $R_1,R_2,\cdots,R_n$，其相应的关键字分别为 $K_1,K_2,\cdots,K_n$；其输出的内容为 $R_{i1},R_{i2},\cdots,R_{in}$，使得 $K_{i1}\leqslant K_{i2}\leqslant\cdots\leqslant K_{in}$ (或 $K_{i1}\geqslant K_{i2}\geqslant\cdots\geqslant K_{in}$)。

1. 排序的对象

排序的对象是文件，它由一组记录组成。每条记录则由一个或若干个数据项(或域)组成。

2. 排序运算的依据

所谓关键字项就是可用来标识一个记录的一个或多个组合的数据项。该数据项的值称为关键字(Key)。需注意的是，在不易产生混淆时，可将关键字项简称为关键字。用来作为排序运算依据的关键字，可以是数字类型，也可以是字符类型。关键字的选取应根据问题的要求而定。

例如，在学生成绩统计中将每个学生作为一个记录。每条记录包含学号、姓名、各科的分数和总分数等内容。若要惟一地标识一个考生的记录，则必须用“学号”作为关键字。若要按照考生的总分数排名次，则需用“总分数”作为关键字。

7.1.2 排序的稳定性

当待排序记录的关键字均不相同时，排序结果是惟一的，否则排序结果不惟一。

在待排序的文件中，若存在多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是稳定的；若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是不稳定的。排序算法的稳定性是针对所有输入实例而言的。即在所有可能的输入实例中，只要有一个实例使得算法不满足稳定性要求，则该排序算法就是不稳定的。

7.1.3 排序的分类

按在排序过程中是否涉及数据的内、外存交换来分类，排序大致分为两类：内部排序和外部排序。在排序过程中，若整个文件都放在内存中处理，排序时不涉及数据的内、外存交换，则称之为内部排序(简称内排序)；反之，若排序过程中要进行数据的内、外存交换，则称之为外部排序。一般情况下，内排序适宜在记录个数不多的小文件中使用，外排序则适用于记录个数太多，不能一次将其全部记录放入内存的大文件。

对于外排序，可以进一步分为两种方法：

- 合并排序法
- 直接合并排序法

对于内排序，按策略进行划分，可以分为：

- 插入排序
- 选择排序
- 交换排序
- 归并排序
- 分配排序

在后面的章节中将分别介绍内排序的几种排序方法和合并排序法。

7.1.4　排序算法分析

当比较两个排序算法时，最直截了当的方法是对它们进行编程，然后比较它们的运行时间。但是，有些算法的运行时间依赖于原始输入记录的情况，特别是记录的数量、记录的大小、关键字的可操作区域以及输入记录的原始有序程度等，这些都会大大影响排序算法的相对运行时间。因此，这种比较方法也就失去了意义。

分析排序算法时，传统方法是衡量关键字之间进行比较的次数。这种方法通常与算法消耗的时间有关，而与机器和数据类型无关。但是在一些情况下，记录也许很大，以致于它们的移动是影响程序整个运行时间的重要因素。因此，排序算法分析应该考虑比较的次数和数据移动的次数。

并不总是需要或是可能确定比较的准确次数，因此只能计算一个近似值。比较和移动的次数都用大 0 表示法，通过给定这些数的数量级来近似。但是，数量级因数据的初始顺序不同而不同。例如，在数据已经排序的情况下，机器需要多少时间用于数据排序呢？它是直接识别出这个初始的排序还是对此毫无觉察呢？因此，效率的测定也指示算法的聪明程度。也正是如此，计算以下 3 种情况下的比较和移动次数：最好情况(通常是数据已经排序)、最坏情况(通常是数据按反序存放)和平均情况(数据是随机顺序的)。有些排序算法无视初始排序的数据，总是执行相同的操作。这些算法的效率是很容易测量的，但是它的性能通常不会很好。有很多排序方法在这 3 种情况下的性能是迥然不同的。比较次数和移动次数不必相同。一个算法可以在前者上非常高效而在后者上非常低效，反之亦然。所以可以根据实际条件选择使用哪一种算法。

7.2　插 入 排 序

插入排序的基本思想是：每次将一个待排序的记录按其关键字大小插入到前面已经排好序的子文件的适当位置，直到全部记录插入完成为止。本节介绍两种插入排序方法：直接插入排序和希尔排序。

7.2.1　直接插入排序

1. 基本思想

直接插入排序是一种最简单的排序方法，它的基本操作是将一个记录插入到已排好序的有序表中，从而得到一个新的有序表。它的基本思想是：假设待排序的记录存放在数组 $R[1\cdots\cdots n]$中，排序过程中，R 被分成两个子区间 $R[1\cdots\cdots i]$和 $R[i+1\cdots\cdots n]$，其中，$R[1\cdots\cdots i]$是已经排好序的有序区；$R[i+1\cdots\cdots n]$是当前未排序的部分。将当前无序区的第一个记录 $R[i+1]$

插入到有序区 $R[1……i]$的适当位置，使 $R[1……i+1]$变为新的有序区，每次插入一个数据，直到所有的数据有序为止。

2. 插入算法

前提条件：序列 $s=\{s_0,s_1,s_2,\cdots,s_{n-1}\}$是 n 个可排序元素的序列。

(1) 令 i 从 1 递增到 $n-1$，重复步骤(2)~(4)。

(2) 将元素 s_i 保存到临时变量中。

(3) 确定使得条件 $s_j \geqslant s_i$ 成立的最小的 j。

(4) 将子序列$\{s_j,\cdots,s_{i-1}\}$后移一个位置到$\{s_{j+1},\cdots,s_i\}$。

(5) 将保存在临时变量中的原来的 s_i 复制到 s_j。

(6) 打印排序结果。

3. Java 程序

按照直接插入排序的算法，其具体程序如下：

```
//=====================Program Description=====================
    //程序名称:insertsort.java
    //程序目的:使用直接插入排序法设计一个排序程序
    //=============================================================
import java.util.*;
import java.io.*;

public class insertsort
{
    public static int[] Data=new int[10];//数据数组

    public static void main (String args[])
    {
        int i;          //循环变量
        int Index;   //数组下标变量

System.out.println("Please input the values you want to sort(Exit for 0):");

        Index=0;//数组下标变量初始值
 //读取输入数据存入数组中
    InputStreamReader is=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(is);
    StringTokenizer st;

            do       //读取输入值
            {
                System.out.print("Data "+Index+":");
```

```
    try{
      String myline=br.readLine();
      st=new StringTokenizer(myline);
Data[Index]=Integer.parseInt(st.nextToken());//取得输入值
    }
catch(IOException ioe)
    {
      System.out.print("IO error:"+ioe);
    }
        Index++;
}while (Data[Index-1] !=0);
                //排序前数据内容
System.out.print("Before Insert Sorting:");
for (i=0;i<Index-1;i++)
      System.out.print(" "+Data[i]+" ");
System.out.println("");

InsertSort(Index-1); //插入排序
                        //排序后结果
System.out.print("After Insert Sorting:");
for (i=0;i<Index-1;i++)
          System.out.print(" "+Data[i]+" ");
      System.out.println("");
    }

//-----------------------------------------------------
//直接插入排序
//-----------------------------------------------------

    public static void InsertSort(int Index)
    {
        int i,j,k;     //循环变量
        int InsertNode;   //欲插入数据变量
for (i=1;i<Index; i++);   //依序插入数值
{
    InsertNode=Data[i]; //设定欲插入的数值
j=i-1;
              //找适当的插入位置
while (j>=0 && InsertNode<Data[j])
{
    Data[j+1]=Data[j];
                  j--;
}
```

```
Data[j+1]=InsertNode;  //将数值插入
        //打印当前排序结果
      System.out.print("Current sorting result:");
      for (k=0;k<Index;k++)
            System.out.print(" "+Data[k]+" ");
System.out.println("");
}
        }
    }
```

4. 直接插入排序法的算法分析

(1) 算法的时间性能分析

对于具有 n 个记录的文件，要进行 $n-1$ 趟排序。

各种状态下的时间复杂度如表 7-1 所示。

表 7-1　各种状态下的时间复杂度

初始文件状态	正　　序	反　　序	无序(平均)
第 i 趟的关键字比较次数	1	$i+1$	$(i-2)/2$
总关键字比较次数	$N-1$	$(n+2)(n-1)/2$	$\approx n^2/4$
第 i 趟记录移动次数	0	$i+2$	$(i-2)/2$
总的记录移动次数	0	$(n-1)(n+4)/2$	$\approx n^2/4$
时间复杂度	$O(n)$	$O(n^2)$	$O(n^2)$

(2) 算法的空间复杂度分析

算法所需的辅助空间是一个监视哨，辅助空间复杂度 $S(n)=O(1)$，是一个就地排序。

(3) 直接插入排序的稳定性

直接插入排序是稳定的排序方法。

7.2.2　希尔排序

1. 基本思想

希尔排序(Shell Sort)是插入排序的一种。因 D.L.Shell 于 1959 年提出而得名。其基本思想是：先取定一个小于 n 的整数 d_1 作为第一个增量，把文件的全部记录分成 d_1 个组，所有距离为 d_1 的倍数的记录放在同一个组中，在各组内进行插入排序；然后，取第二个增量 $d_2<d_1$，重复上述的分组和排序，直至所取的增量 $d_t=1(d_t<d_{t-1}<\cdots<d_2<d_1)$，即所有记录放在同一组中进行直接插入排序为止。

2. 具体算法

按照希尔排序的基本思想，其具体算法如下：

```
//=====================Program Description=====================
//程序名称:shellsort.java
//程序目的:使用希尔排序法设计一个排序程序
//=============================================================
import java.util.*;
import java.io.*;
public class shellsort
{
    public static int[] Data=new int[20];//数据数组
    public static void main (String args[])
    {
        int i;//循环变量
        int Index;//数组下标变量
System.out.println("Please input the values you want to sort(Exit for 0):");
        Index=0;//数组下标变量初始值
        //读取输入数据存入数组中
            InputStreamReader is=new InputStreamReader(System.in);
            BufferedReader br=new BufferedReader(is);
            StringTokenizer st;

            do      //读取输入值
{
        System.out.print("Data "+Index+":");
            try{
            String myline=br.readLine();
            st=new StringTokenizer(myline);
            Data[Index]=Integer.parseInt(st.nextToken());//取得输入值
           }
        catch(IOException ioe)
           {
            System.out.print("IO error:"+ioe);
           }
        Index++;
    }while (Data[Index-1] !=0);
                                //排序前数据内容
System.out.print("Before Shell Sorting:");
for (i=0;i<Index-1;i++)
        System.out.print(" "+Data[i]+" ");
System.out.println("");
ShellSort(Index-1);//希尔排序
```

```
            System.out.print("After Shell Sorting:");
            for (i=0;i<Index-1;i++)
                System.out.print(" "+Data[i]+" ");
            System.out.println("");
        }
//-----------------------------------------------------
//希尔排序
//-----------------------------------------------------
public static void ShellSort(int Index)
{
    int i,j,k;                                  //循环变量
    int Temp;                                   //暂存变量
    boolean  Change;                            //数据是否改变
    int DataLength;                             //分割集合的间隔长度
    int Pointer;                                //进行处理的位置
    DataLength=(int) Index/2;                   //初始集合间隔长度
    while (DataLength !=0)                      //数列仍可进行分割
    {                                           //对各个集合进行处理
        for(j=DataLength;j<Index;j++)
        {
        Change=false;
         Temp=Data[j];                          //暂存 Data[j]的值，待交换值时用
          Pointer=j-DataLength;                 //计算进行处理的位置
                    //进行集合内数值的比较与交换值
while (Temp<Data[Pointer] && Pointer>=0 && Pointer<=Index)
            {
                Data[Pointer+DataLength]=Data[Pointer];
                    //计算下一个欲进行处理的位置
                Pointer=Pointer-DataLength;
                Change=true;
                if(Pointer<0 || Pointer>Index)
                    break;
            }
                        //与最后的数值交换
            Data[Pointer+DataLength]=Temp;
          if(Change)
            {                                   //打印目前排序结果
                System.out.print("Current sorting result:");
                for (k=0;k<Index;k++)
                    System.out.print(" "+Data[k]+" ");
            System.out.println("");
        }                       }
    DataLength=DataLength/2;     //计算下次分割的间隔长度
}}}
```

3. 希尔排序的算法分析

(1) 增量序列的选择

Shell 排序的执行时间依赖于增量序列。

好的增量序列的共同特征为：

- 最后一个增量必须为 1；
- 应该尽量避免序列中的值(尤其是相邻的值)互为倍数的情况。

通过大量的实验，给出了目前较好的结果：当 n 较大时，比较和移动的次数约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 之间。

(2) Shell 排序的时间性能优于直接插入排序

希尔排序的时间性能优于直接插入排序的原因如下：

- 当文件初态基本有序时直接插入排序所需的比较和移动次数均较少。
- 当 n 值较小时，n 和 n^2 的差别也较小，即直接插入排序的最好时间复杂度 $O(n)$和最坏时间复杂度 $O(n^2)$差别不大。
- 在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量 d_i 逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多，但由于已经按 d_{i-1} 作为距离排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快。因此，希尔排序在效率上较直接插入排序有较大的改进。

(3) 稳定性

希尔排序是一种不稳定的排序方法。

7.3 交 换 排 序

交换排序的基本思想是：两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。应用交换排序基本思想的主要排序方法有冒泡排序和快速排序。

7.3.1 冒泡排序

1. 基本思想

冒泡排序的基本思想是：设想被排序的记录关键字保存在数组 $R[1\cdots n]$中，将每个记录 $R[i]$看作是重量为 $R[i]$.key 的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R；凡扫描到违反本原则的轻气泡，就使其向上“飘浮”。如此反复进行，直到最后任何两个气泡都是轻者在上，重者在下为止。

2. 具体算法

前提条件：序列 $s=\{s_0,s_1,s_2,\ldots,s_{n-1}\}$是 n 个可排序元素的序列。

(1) 令 j 从 $n-1$ 递减到 1，重复步骤(2)~(4)。

(2) 令 i 从 1 递增到 j，重复步骤(3)。

(3) 如果元素 s_{i-1} 和 s_i 成反序，交换它们。

(4) 结束标记，序列$\{s_0,\cdots,s_j\}$被排序且 s_j 最大。

3. Java 程序

按照冒泡排序的算法，其具体程序如下：

```
// ==========Program Descrition==========
// 程序名称：bubblesort.java
// 程序目的：使用冒泡排序法设计一个排序程序。
// ======================================
import java.util.*;
import java.io.*;
public class bubblesort
{
 public static int[] Data = new int[10];
    public static void main (String args[])
    {
      int i;          //循环计数变量
      int Index;   //数组下标变量
System.out.println("Please input the values you want to sort (Exit for 0):");
      Index = 0;
         //读取输入数据存入数组中
            InputStreamReader is=new InputStreamReader(System.in);
            BufferedReader br=new BufferedReader(is);
            StringTokenizer st;
      do              //读取输入值
      {
           System.out.print("Data "+Index+" : ");
             try{
              String myline=br.readLine();
              st=new StringTokenizer(myline);
              Data[Index]=Integer.parseInt(st.nextToken());//取得输入值
            }
    catch(IOException ioe)
        {
         System.out.print("IO error:"+ioe);
        }
```

```
        Index++;
}while ( Data[Index-1] != 0);                          //排序前数据内容
System.out.print("Before Bubble Sorting :");
for ( i=0 ; i<Index-1 ; i++ )
        System.out.print(" "+Data[i]+" ");
System.out.println("");
BubbleSort(Index-1); //冒泡排序
System.out.print("After Bubble Sorting : ");            //排序后结果
        for ( i=0 ; i<Index-1 ; i++ )
        System.out.print(" "+Data[i]+" ");
    System.out.println("");
 }
//-------------------------------------------------
//冒泡程序
//-------------------------------------------------
        public static void BubbleSort(int Index)
    {
        int i,j,k;                                     //循环变量
        boolean Change;                                //数据是否有改变
        int Temp;                                      //数据暂存变量
        for ( j=Index ; j>1 ; j-- )                    //外层循环
{
    Change = false;                                    //设置为数据未改变
  for ( i=0 ; i<j-1 ; i++ )                            //内层循环
  {         if (Data[i+1] < Data[i] )                  //比较两数值
      {
          Temp = Data[i+1];
          Data[i+1] = Data[i];
          Data[i]   =Temp;
          Change = true;   //设置数据已改变
      }                          }
if ( Change ) //如果数据已改变则输出结果
{ System.out.print("Current Sorting Result : ");
                                   //打印目前排序状况
                    for ( k=0 ; k<Index ; k++ )
                        System.out.print(" "+Data[k]+" ");
                    System.out.println("");
                }
        }}}
```

4. 冒泡排序的算法分析

(1) 算法的最好时间复杂度

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数 C 和记录移动次数 M 均达到最小值：

$$C_{\min}=n-1$$
$$M_{\min}=0$$

冒泡排序最好的时间复杂度为 $O(n)$。

(2) 算法的最坏时间复杂度

若初始文件是反序的，需要进行 $n-1$ 趟排序。每趟排序要进行 $n-i$ 次关键字的比较($1\leqslant i\leqslant n-1$)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：

$$C_{\max}=n(n-1)/2=O(n^2)$$
$$M_{\max}=3n(n-1)/2=O(n^2)$$

冒泡排序的最坏时间复杂度为 $O(n^2)$。

(3) 算法的平均时间复杂度为 $O(n^2)$

虽然冒泡排序不一定要进行 $n-1$ 趟，但由于它的记录移动次数较多，故平均时间性能比直接插入排序要差得多。

(4) 算法稳定性

冒泡排序是就地排序，且它是稳定的。

5. 冒泡排序的算法改进

上述的冒泡排序还可作如下的改进。

(1) 记住最后一次交换发生位置 lastExchange 的冒泡排序

在每趟扫描中，记住最后一次交换发生的位置 lastExchange (该位置之前的相邻记录均已有序)。下一趟排序开始时，R[1……lastExchange－1]是有序区，R[lastExchange……n]是无序区。这样，一趟排序可能使当前有序区扩充多个记录，从而减少排序的趟数。

(2) 改变扫描方向的冒泡排序

① 冒泡排序的不对称性

能一趟扫描完成排序的情况：只有最轻的气泡位于 $R[n]$的位置，其余的气泡均已排好序，那么也只需一趟扫描就可以完成排序。例如，对初始关键字序列 12，18，42，44，45，67，94，10 就仅需一趟扫描。

需要 $n-1$ 趟扫描完成排序情况：当只有最重的气泡位于 R[1]的位置，其余的气泡均已排好序时，则仍需做 $n-1$ 趟扫描才能完成排序。例如，对初始关键字序列 94，10，12，18，42，44，45，67 就需 7 趟扫描。

② 造成不对称性的原因

每趟扫描仅能使最重气泡“下沉”一个位置，因此使位于顶端的最重气泡下沉到底部时，需作 $n-1$ 趟扫描。

③ 改进不对称性的方法

在排序过程中交替改变扫描方向，可改进不对称性。

7.3.2　快速排序

1. 基本思想

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题，递归地解这些子问题。然后将这些子问题的解组合为原问题的解。因此在用递归描述的分治算法的每一层递归上，都有如下 3 个步骤。

步骤 1，分解：将原问题分解为若干个子问题，此步骤亦称为划分；

步骤 2，求解：递归地解各子问题，若子问题的规模足够小，则直接求解；

步骤 3，组合：将各子问题的解组合成原问题的解。

设当前待排序的无序区为 *R*[low……high]，利用分治法可将快速排序的基本思想描述为：

(1) 分解。在 *R*[low……high]中任选一个记录作为基准，以此基准将当前无序区划分为左、右两个较小的子区间 *R*[low……pivotpos－1]和 R[pivotpos+1……high]，并使左边子区间中所有记录的关键字均小于等于基准记录(不妨记为 pivot)的关键字 pivot.key，右边的子区间中所有记录的关键字均大于等于 pivot.key，而基准记录 pivot 则位于正确的位置(pivotpos)上，它无须参加后续的排序。因此，划分的关键是要求出基准记录所在的位置 pivotpos，划分的结果可以简单地表示为(注意 pivot=R[pivotpos])：

R[low……pivotpos-1]).keys≤R[pivotpos].key≤R[pivotpos+1……high].key

这里 low≤pivotpos≤high。

(2) 求解。通过递归调用快速排序对左、右子区间 R[low……pivotpos-1] 和 R[pivotpos+1……high]排序。

(3) 组合。因为当“求解”步骤中的两个递归调用结束时，其左、右两个子区间已有序，所以由上面的不等式立即知道整个数组 *R* 已有序。

2. 具体算法

按照快速排序的基本思想，其具体算法如下：

```
// ═══════════Program Descrition═══════════
// 程序名称：quicksort.java
// 程序目的：使用快速排序法设计一个排序程序。
// ═══════════════════════════════════════
import java.util.*;
```

```
import java.io.*;
public class quicksort
{
 public static int[] Data = new int[10];
     public static void main (String args[])
     {
       int i;          //循环变量
       int Index;   //数组下标变量
System.out.println("Please input the values you want to sort (Exit for 0):");
       Index = 0;
          //读取输入数据存入数组中
              InputStreamReader is=new InputStreamReader(System.in);
              BufferedReader br=new BufferedReader(is);
              StringTokenizer st;
       do             //读取输入值
       {
             System.out.print("Data "+Index+" : ");
               try{
                  String myline=br.readLine();
                  st=new StringTokenizer(myline);
                  Data[Index]=Integer.parseInt(st.nextToken());//取得输入值
               }
            catch(IOException ioe)
               {
                System.out.print("IO error:"+ioe);
               }

       Index++;
}while ( Data[Index-1] != 0);
                                          //排序前数据内容
System.out.print("Before Quick Sorting :");
for ( i=0 ; i<Index-1 ; i++ )
       System.out.print(" "+Data[i]+" ");
System.out.println("");
       QuickSort(0,Index-2,Index-1); //快速排序
                                       //排序后结果
       System.out.print("After Bubble Sorting : ");
       for ( i=0 ; i<Index-1 ; i++ )
             System.out.print(" "+Data[i]+" ");
       System.out.println("");
 }
//-------------------------------------------------
//快速排序程序
```

```
//-------------------------------------------------
        public static void QuickSort(int Left,int Right,int Index)
    {
        int i,j,k;          //循环变量
        int Pivot;          //枢纽变量
        int Temp;           //数据暂存变量
      i= Left;
      j= Right-1
      Pivot= Data[Left]
       If (i<j)
      {
       do
       {
        do
        {
         i++;
        } while (Data[i]<=Pivot && i<=Right);//从左向右找比 Pivot 大的值
        do
        {
         i--;
        } while (Data[i]>=Pivot && i>=Left); //从右向左找比 Pivot 小的值
       if (i<j)
       {
        Temp=Data[i];
        Data[i]= Data[j];
        Data[j]=Temp;
       }
      } while (i<j);//交换 Data[i]和 Data[j]的值
       if (i>j)
       {
        Temp=Data[Left]; //交换 Data[Left]和 Data[j]的值
        Data[Left]= Data[j];
        Data[j]=Temp;
                        //打印目前排序状况
          System.out.print("Current Sorting Result : ");
          for ( k=0 ; k<Index ; k++ )
          System.out.print(" "+Data[k]+" ");
          System.out.println("");
          }
       QuickSort(Left,j-1,Index);    //排序左边
       QuickSort(j+1,right,Index);   //排序右边
}}}
```

3. 算法分析

快速排序的时间主要耗费在划分操作上，对长度为 k 的区间进行划分，共需 $k-1$ 次关键字的比较。

(1) 最坏时间复杂度

最坏情况是每次划分选取的基准都是当前无序区中关键字最小(或最大)的记录，划分的结果是基准左边的子区间为空(或右边的子区间为空)，而划分所得的另一个非空的子区间中记录数目仅仅比划分前的无序区中记录个数减少一个。

因此，快速排序必须做 $n-1$ 次划分，第 i 次划分开始时区间长度为 $n-i+1$，所需的比较次数为 $n-i(1\leqslant i\leqslant n-1)$，故总的比较次数达到最大值：

$$C_{max}=n(n-1)/2=O(n^2)$$

如果按上面给出的划分算法，每次取当前无序区的第 1 个记录为基准，那么当文件的记录已按递增序(或递减序)排列时，每次划分所取的基准就是当前无序区中关键字最小(或最大)的记录，则快速排序所需的比较次数反而最多。

(2) 最好时间复杂度

在最好情况下，每次划分所取的基准都是当前无序区的“中值”记录，划分的结果是基准的左、右两个无序子区间的长度大致相等。总的关键字比较次数为：

$O(n\lg n)$

注意：

用递归树来分析最好情况下的比较次数更简单。因为每次划分后，左、右子区间长度大致相等，故递归树的高度为 $O(\lg n)$，而递归树每一层上各结点对应的划分过程中所需要的关键字比较次数总和不超过 n，故整个排序过程所需要的关键字比较总次数 $C(n)=O(n\lg n)$。

因为快速排序的记录移动次数不大于比较的次数，所以快速排序的最坏时间复杂度应为 $O(n^2)$，最好时间复杂度为 $O(n\lg n)$。

(3) 基准关键字的选取

在当前无序区中选取划分的基准关键字是决定算法性能的关键。

① “三者取中”的规则

“三者取中”规则，即在当前区间里，将该区间首、尾和中间位置上的关键字比较，取三者的中值所对应的记录作为基准，在划分开始前将该基准记录和该区间的第 1 个记录进行交换，此后的划分过程与上面所给的 Partition 算法完全相同。

② 取位于 low 和 high 之间的随机数 $k(low\leqslant k\leqslant high)$，用 $R[k]$作为基准。

选取基准最好的方法是用一个随机函数产生一个位于 low 和 high 之间的随机数 $k(low\leqslant k\leqslant high)$，用 $R[k]$作为基准，这相当于强迫 $R[low\cdots\cdots high]$中的记录是随机分布的。用此方法所得到的快速排序一般称为随机的快速排序。

注意：

随机化的快速排序与一般的快速排序算法差别很小。但随机化后，算法的性能大大地提高了，尤其是对初始有序的文件，一般不可能导致最坏情况的发生。算法的随机化不仅仅适用于快速排序，也适用于其他需要数据随机分布的算法。

(4) 平均时间复杂度

尽管快速排序的最坏时间为 $O(n^2)$，但就平均性能而言，它是基于关键字比较的内部排序算法中速度最快者，快速排序亦因此而得名。它的平均时间复杂度为 $O(n\lg n)$。

(5) 空间复杂度

快速排序在系统内部需要一个栈来实现递归。若每次划分较为均匀，则其递归树的高度为 $O(\lg n)$，故递归后需栈空间为 $O(\lg n)$。最坏情况下，递归树的高度为 $O(n)$，所需的栈空间为 $O(n)$。

(6) 稳定性

快速排序是非稳定的。

7.4 选 择 排 序

选择排序(Selection Sort)的基本思想是：每一趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子文件的最后，直到全部记录排序完毕。主要有两种选择排序方法：直接选择排序(或称简单选择排序)和堆排序。

7.4.1 直接选择排序

1. 基本思想

直接选择排序的基本思想是：第 i 趟排序开始时，当前有序区和无序区分别为 $R[1\cdots\cdots i-1]$和 $R[i\cdots\cdots n](1\leqslant i\leqslant n-1)$，该趟排序则是从当前无序区中选出关键字最小的记录 $R[k]$，将它与无序区的第 1 个记录 $R[i]$交换，使 $R[1\cdots\cdots i]$和 $R[i+1\cdots\cdots n]$分别变为新的有序区和新的无序区。因为每趟排序均使有序区中增加了一个记录，且有序区中的记录关键字均不大于无序区中记录的关键字，即第 i 趟排序之后 $R[1\cdots\cdots i]$.keys$\leqslant R[i+1\cdots\cdots n]$.keys，所以进行 $n-1$ 趟排序之后有 $R[1\cdots\cdots n-1]$.keys$\leqslant R[n]$.key，即经过 $n-1$ 趟排序之后，整个文件 $R[1\cdots\cdots n]$递增有序。注意，第 1 趟排序开始时，无序区为 $R[1\cdots\cdots n]$，有序区为空。

2. 具体算法

按照直接选择排序的基本思想，其具体算法如下：

```
//=====================Program Description=====================
//程序名称:selectsort.java
//程序目的:使用选择排序法设计一个排序程序
//=============================================================
import java.util.*;
import java.io.*;
public class selectsort
{
    public static int[] Data=new int[10];
    public static void main (String args[])
    {
        int i;    //循环计数变量
        int Index; //数组下标变量
System.out.println("Please input the values you want to sort(Exit for 0):");
        Index=0;//数组下标变量初始值
 //读取输入数据存入数组中
    InputStreamReader is=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(is);
    StringTokenizer st;
            do
            {
            System.out.print("Data "+Index+":");
            try{
                String myline=br.readLine();
                st=new StringTokenizer(myline);
            Data[Index]=Integer.parseInt(st.nextToken());//取得输入值
                }
             catch(IOException ioe)
                {
                 System.out.print("IO error:"+ioe);
                }

            Index++;
}while (Data[Index-1] !=0);
                    //排序前数据内容
System.out.print("Before Select Sorting:");
for (i=0;i<Index-1;i++)
      System.out.print(" "+Data[i]+" ");
System.out.println("");
SelectSort(Index-1);//选择排序
                        //排序后结果
System.out.print("After Select Sorting:");
for (i=0;i<Index-1;i++)
```

```
                System.out.print(" "+Data[i]+" ");
            System.out.println("");
        }
//-----------------------------------------------------
//直接选择排序
//-----------------------------------------------------
        public static void SelectSort(int Index)
        {
            int i,j,k;                  //循环计数变量
            int MinValue;               //最小值变量
            int IndexMin;               //最小值下标变量
            int Temp;                   //暂存变量
            for (i=0;i<Index-1;i++)
            {
                MinValue=32767;   //目前最小值
                IndexMin=0;         //存储最小值的数组下标
                for(j=i;j<Index;j++)
                {
                    if(Data[j]<MinValue)   //找到最小值
                        {
                            MinValue=Data[j]; //存储最小值
                            IndexMin=j;
                            }
                            Temp=Data[i];               //交换两数值
                            Data[i]=Data[IndexMin];
                            Data[IndexMin]=Temp;
                }
                System.out.print("Current sorting result:");
                for (k=0;k<Index;k++)
                    System.out.print(" "+Data[k]+" ");
                System.out.println("");
}}}
```

3. 直接选择排序的算法分析

(1) 关键字比较次数

无论文件初始状态如何，在第 i 趟排序中选出最小关键字的记录，需做 $n-i$ 次比较，因此，总的比较次数为：

$$n(n-1)/2=O(n^2)$$

(2) 记录的移动次数

当初始文件为正序时，移动次数为 0。文件初态为反序时，每趟排序均要执行交换操

作，总的移动次数取最大值 $3(n-1)$。

直接选择排序的平均时间复杂度为 $O(n^2)$。

(3) 直接选择排序是一个就地排序

(4) 稳定性分析

直接选择排序是不稳定的。

7.4.2　堆排序

1. 基本思想

堆排序是利用完全二叉树进行排序的方法。

堆首先是一棵完全二叉树，然后满足以下条件之一：

(1) $K_i \leqslant K_{2i}$　　并且 $K_i \leqslant K_{2i+1}$

(2) $K_i \geqslant K_{2i}$　　并且 $Ki \geqslant K_{2i+1}$

堆有大根堆(根结点的关键字值最大的堆)和小根堆(根结点关键字值最小)之分。

堆排序利用了大根堆(或小根堆)堆顶记录的关键字最大(或最小)这一特征，使得在当前无序区中选取最大(或最小)关键字的记录变得简单。假设使用大根堆进行排序，其基本思想是：首先将初始文件 $R[1\cdots\cdots n]$建成一个大根堆，此堆为初始的无序区；将关键字最大的记录 $R[1]$(即堆顶)和无序区的最后一个记录 $R[n]$交换，由此得到新的无序区 $R[1\cdots\cdots n-1]$和有序区 $R[n]$，且满足 $R[1\cdots\cdots n-1]$.keys$\leqslant R[n]$.keys，由于交换后新的根 $R[1]$可能违反堆性质，故应将当前无序区 $R[1\cdots\cdots n-1]$调整为堆；然后再次将 $R[1\cdots\cdots n-1]$中关键字最大的记录 $R[1]$和该区间的最后一个记录 $R[n-1]$交换，由此得到新的无序区 $R[1\cdots\cdots n-2]$和有序区 $R[n-1\cdots\cdots n]$，且仍满足关系 $R[1\cdots\cdots n-2]$.keys$\leqslant R[n-1\cdots\cdots n]$.keys。同样要将 $R[1\cdots\cdots n-2]$调整为堆。重复以上步骤，直至按关键字有序。

由此可抽象出大根堆排序算法的基本操作：初始化操作是将 $R[1\cdots\cdots n]$构造为初始堆；每一趟排序的基本操作是将当前无序区的堆顶记录 $R[1]$和该区间的最后一个记录交换，然后将新的无序区调整为堆(亦称重建堆)。显然，只需要做 $n-1$ 趟排序，选出较大的 $n-1$ 个关键字即可使文件递增有序。用小根堆排序完全与此类同，只不过其排序结果是递减有序的。

2. 具体算法

按照堆排序的基本思想，其具体算法如下：

```
//=================Program Description=============================
//程序名称:Heapsort.java
//程序目的:使用堆排序法设计一个排序程序
//=================================================================
import java.util.*;
```

```
import java.io.*;
public class heapsort
{
    public static int[] Heap=new int[10];//堆数组
    public static void main(String args[])
    {
      int i;//循环变量
 int Index;//数组下标变量
system.out.println("Please input the values you want to sort(Exit for 0): ");
    Index=1;                                    //读取输入数据存入数组中
    InputStreamReader is=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(is);
    StringTokenizer st;
     do        //读取输入值
     {
       System.out.print("Data "+Index+":");
       try{
                String myline=br.readLine();
                st=new StringTokenizer(myline);
                Heap[Index]=Integer.parseInt(st.nextToken());//取得输入值
                }
              catch(IOException ioe)
                {
                 System.out.print("IO error:"+ioe);
                }
              Index++;
}while (Heap[Index-1] !=0);
                                    //排序前数据内容
System.out.print("Before Heap Sorting: ");
for (i=1;i<Index-1;i++)
     System.out.print(""+Heap[i]+ "");
System.out.println("");
HeapSort(Index-2);
                                    //排序后数据内容
          System.out.print("After Heap Sorting: ");
          for (i=1;i<Index-1;i++)
               System.out.print(""+Heap[i]+ "");
          System.out.println("");
          }
//=============================================================
//建立堆
//=============================================================
          public static void CreateHeap(int Root,int Index)
```

```
{
   int i,j;//循环变量
   int Temp;//暂存变量
   int Finish;//判断是否完成
   j=2*Root;//子节点的 Index
   Temp=Heap[Root];//暂存堆的 Root 值
   Finish=0;//预设堆尚未完成
While (j<=Index && Finish ==0)
{
   if (j<Index)   //找最大的子节点
      if (Heap[j]<Heap[j+1])
         j++;
   if (Temp>=Heap[j])
         Finish=1;//堆建立完成
    else
       {
         Heap[j/2]=Heap[j];   //父节点=目前节点
         j=2*j;
        } }
    Heap[j/2]=Temp;   //父节点=Root 值
          }
//=============================================
//堆排序
//=============================================
          public static void HeapSort(int Index)
          {
             int i,j,Temp;
                                  //将二叉树转成堆
             for (i=(Index/2),i>=1;i--)
                CreateHeap(i,Index);
                                        //开始进行堆排序
             for (i=(Index-1),i>=1;i--)
             {
                Temp=Heap[i+1];   //堆的 Root 值和最后一个值交换
                Heap[i+1]=Heap[1];
                Heap[1]=Temp;
                CreateHeap(1,i);//对其余数值重建堆
//打印堆的处理过程
                System.out.print("Sorting Processing: ");
                for (j=1;j<=Index;j++)
                   System.out.print(""+Heap[j]+"");
                System.out.println("");
             } }}
```

3. 算法分析

堆排序的时间主要由建立初始堆和反复重建堆这两部分的时间开销构成，它们均是通过调用 HeapSort 实现的。

堆排序的最坏时间复杂度为 $O(n\lg n)$。堆排序的平均性能较接近于最坏性能。

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

堆排序是就地排序，辅助空间为 $O(1)$，它是不稳定的排序方法。

堆排序的时间复杂度为 $O(n\lg n)$，是一种不稳定的排序方法。

7.5 归 并 排 序

1. 基本思想

归并排序是将两个或两个以上的有序表组合成一个新的有序表。其基本思想是：先将 N 个数据看成 N 个长度为 1 的表，将相邻的表成对合并，得到长度为 2 的 $N/2$ 个有序表，进一步将相邻的合并，得到长度为 4 的 $N/4$ 个有序表，以此类推，直到所有数据均合并成一个长度为 N 的有序表为止。每一次归并过程称做一趟。

要解决归并问题，首先需要解决两两归并问题(两个有序表合并成一个有序表)。其 Java 程序为：

```
//-----------------------------------------------------
//两两归并
//-----------------------------------------------------
    public static void MergeTwo(int Left,int Middle,int N)
    {       int i,j,k,t;   //循环变量
             i=Left;
             k=Left;
             j=Middle+1;   //设定数组指针
      while (i<=Middle && j<=N) //两个欲合并的数组均还有值尚未处理
           {
        if (Data[i]<=Data[j]) //将较小者先存到输出数组
        { Output[k]=Data[i];
          i++;
        }
        else
        {   Output[k]=Data[j];
             j++;
        }
        k=k+1;
    }     //将尚未处理完的数组数值依序存入输出数组
```

```
if (i>Middle)
{       for (t=j;t<=N;t++)
        Output[k+t-j]=Data[t];
}
else
{   for (t=i;t<=Middle;t++)
    Output[k+t-j]=Data[t];
} }
```

2. 实现方法

归并排序有两种实现方法：自底向上和自顶向下。自底向上的基本思想是：第 1 趟归并排序时，将待排序的文件 $R[1\cdots\cdots n]$看作是 n 个长度为 1 的有序子文件，将这些子文件两两归并，若 n 为偶数，则得到 $n/2$ 个长度为 2 的有序子文件；若 n 为奇数，则最后一个子文件轮空(不参与归并，直接并入下一趟归并)，故本趟归并完成后，前 $n/2-1$ 个有序子文件长度为 2，但最后一个子文件长度仍为 1；第 2 趟归并则是将第 1 趟归并所得到的 $n/2$ 个有序的子文件两两归并，如此反复，直到最后得到一个长度为 n 的有序文件为止。

以上的算法也称为二路归并。但它只是一次合并，要完成一趟归并，需要重复调用上述过程。

其程序实现为：

```
import java.io.*;
public class mergesort
{   public static int[] Data=new int[10];        //预设数据数组
     public static int[] Output=new int[10];   //输出数据数组
     public static void main (String args[])
     {     int i;              //循环变量
           int Index;          //数组下标变量
           int DataDataLength;
           System.out.println("Please input the values you want to sort(Exit for 0):");
           Index=0;          //数组下标变量初始值
                             //读取输入数据存入数组中
              InputStreamReader is=new InputStreamReader(System.in);
           BufferedReader br=new BufferedReader(is);
           StringTokenizer st;
            do   //读取输入值
            {     System.out.print("Data "+Index+":");
                  try{   String myline=br.readLine();
                         Data[Index]=Integer.parseInt(myline);//取得输入值
                             }
                         catch(IOException ioe)
                             { System.out.print("IO error:"+ioe);
```

```
                    }
                Index++;
        }while (Data[Index-1]!=0);
                    //排序前数据内容
        System.out.print("Before Merge Sorting:");
        for (i=0;i<Index-1;i++)
            System.out.print(" "+Data[i]+" ");
        System.out.println("");
        DataDataLength=1;
        while (DataDataLength < Index)
        {    System.out.println("Merge Sort DataLength:"+DataDataLength) ;
             MergeAll(Index-2,DataDataLength); //归并排序
             DataDataLength=2*DataDataLength;
        }     //排序后结果
        System.out.print("After Merge Sorting:");
        for (i=0;i<Index-1;i++)
             System.out.print(" "+Data[i]+" ");
        System.out.println("");
    }
//-----------------------------------------------------
//将所有 partition array 分别来两两合并
//-----------------------------------------------------
    public static void MergeAll(int N,int DataLength)
    {    int i,t;
         i=0;
          //还有两段长度为 Datalength 的 list 可合并
         while(i<=(N-2*DataLength+1))
         {    MergeTwo(i,i+DataLength-1,i+2*DataLength-1);
              i=i+2*DataLength;
         }
        if (i+DataLength-1<N)
{   //合并两段 list，一段长度为 DataLength 的 list 中的值依次序存到输出数组 Output
         MergeTwo(i,i+DataLength-1,N);
         }
        else
        {    for(t=i;t<=N;t++)
             Output[t]=Data[t];
        }
  //将 Output 中的值复制到 Data
        for(t=0;t<=N;t++)
            Data[t]=Output[t];
        System.out.print("current sorting result:");
        for(i=0;i<=N;i++)
```

```
            System.out.print(" "+Output[i]+" ");
        System.out.println("");
    }}
```

3. 算法分析

(1) 稳定性

归并排序是一种稳定的排序。

(2) 存储结构要求

可用顺序存储结构，也易于在链表上实现。

(3) 时间复杂度

对长度为 n 的文件，需进行$\lfloor \log_2^n \rfloor$趟二路归并，每趟归并的时间为 $O(n)$，故其时间复杂度无论是在最好情况下还是在最坏情况下均是 $O(n\lg n)$。

(4) 空间复杂度

需要一个辅助向量来暂存两有序子文件归并的结果，故其辅助空间复杂度为 $O(n)$，显然它不是就地排序。

注意：

若用单链表作存储结构，很容易给出就地的归并排序。

7.6 外 部 排 序

算法和数据结构的实现可以基于主存储器，也可以基于辅助存储器(如磁盘和磁带等)，但这会影响算法和数据结构的设计。主存储器和辅助存储器的差别主要与存储介质中的访问速度、数据的存储量和数据的永久性有关。大多数文件处理技术都基于这样一个基本事实：访问辅助存储器比访问主存储器要慢很多，而外部排序的过程需要进行多次的主存储器和辅助存储器之间的交换。下面首先讨论对辅助存储器进行存取的特点。

7.6.1 辅助存储器的存取

1. 磁盘信息的存取

磁盘是一种直接存取的存储设备，它是以存取时间变化不大为特征的。它可以直接存取任何字符组。它的容量大、速度快，存取速度比磁带快很多。磁盘是一个扁平的圆盘，盘面上有许多称为磁道的圆圈，信息就记载在磁道上。由于磁道的圆圈为许多同心圆，所以可以直接存取。磁盘可以是单片的，也可以由若干盘片组成盘组。每一片上有两个面。

磁盘驱动器执行读/写信息的功能。盘片装在一个主轴上，并绕主轴高速旋转，当磁道

在读/写头下通过时，便可以进行信息的读/写。

可以把磁盘分为固定头盘和活动头盘。固定头盘的每一道上都有独立的磁头，它是固定不动的，专负责读/写某一道上的信息。活动头盘的磁头是可移动的。盘组是可变的。一个面上只有一个磁头，它可以从该面上的一道移动到另一道。磁头装在一个动臂上，不同面上的磁头是同时移动的，并处于同一圆柱面上。各个面上半径相同的磁道组成一个圆柱面，圆柱面的个数就是盘片面上的磁道数。

为了访问一块信息，必须首先找到柱面，移动臂使磁头移动到所需柱面上，然后等待要访问的信息转到磁头之下，最后是对所需信息的读/写操作。所以在磁盘上读/写一块信息所需的时间由 3 部分组成：寻查时间(读/写头定位的时间)、等待时间(等待信息块的初始位置旋转到读/写头下的时间)和传输时间。

2. 磁带信息的存取

磁带是涂上薄薄一层磁性材料的窄带。磁带不是连续运转的设备，而是一种启停设备(启停时间约为 5ms)，它可以根据读/写的需要随时启动或停止。一般来说，磁带比磁盘要便宜许多。

磁带和磁盘的不同之处是磁带只能顺序访问。要想从磁带的当前位置到达目的位置，必须正转或反转磁带。这使得对于随机访问情况来说，磁带慢得无法接受。由于读/写信息应在旋转稳定时进行，因而磁带从静止状态启动后，要经过一个加速的过程才能达到稳定状态。同样的道理，读/写操作结束时，从运动状态到完全停止要经过一个减速的过程。这样，就必须使用一个很大的间隔 IRG(Iner Record Gap)分开数据，以便磁头能够识别出一个间隔。但是，在每两条记录之间放置一个间隔就会浪费大量的空间。为了避免这种浪费，需要把多条记录组织到一个块中。这样，每个字符组间就没有 IRG，而变成了块间间隔 IBG(Iner Block Gap)。这样一来，通过成块的方法不仅可以减少 IRG 的数目，还可以减少 I/O 操作。

由于磁带很便宜，速度又慢，而且只适合于顺序访问，所以通常不用它来存储需要快速随机访问的数据。它通常被用于备份和归档。

3. 缓冲技术

由于对辅助存储器进行读/写操作相对于 CPU 的处理速度来说是很慢的，而大多数磁盘控制器能独立于 CPU 进行操作。因此，为了解决 CPU 快而 I/O 慢的问题，操作系统使用了缓冲技术。

(1) 解决信息的到达率和离去率不一致的矛盾。

(2) 缓存起中转站的作用。

(3) 使得一次输入的信息能多次使用。

实现缓冲技术的方式有两种。第 1 种是在通道或控制器内设置数据缓冲寄存器作为专用硬件缓冲器，可暂存 I/O 信息，以减少中断 CPU 的次数；第 2 种是在内存中开辟出专用内存缓冲区，作为软件缓冲。

现在，几乎所有的操作系统都自动进行扇区级缓冲，而且磁盘驱动器的控制器硬件中通常也直接建立扇区级缓冲。大多数操作系统至少维护两个缓冲区，一个缓冲区用于输入，另一个缓冲区用于输出。现在，操作系统或应用程序可以在多个缓冲区中存储信息。存储在一个缓冲区中的信息通常称为一页，这些缓冲区合起来称为缓冲池。缓冲池的目标是增加存储器中存储的信息量，对于新的信息请求，从缓冲池中得到请求信息的可能性更大，而不必再从磁盘中读出。

7.6.2　外部排序的方法

如果操作系统支持虚拟存储，最简单的外部排序方法是把整个文件读入虚拟存储器中，然后运行一个内部排序方法，例如快速排序。如果使用这种方法，虚拟存储管理器就可以使用它的缓冲池机制控制磁盘访问。但是，这种方法不总是可行的。一个潜在的问题是虚拟存储器的大小通常比可用磁盘空间小得多。这样，输入文件可能无法放到虚拟存储器中。如果调整内部排序方法，并结合缓冲管理技术之一，就可以克服虚拟存储器大小的限制。

调整内部排序算法使之应用于外部排序，这种思路的更普遍的问题是这样做不可能比设计一个新的尽量减少磁盘存取的算法更有效。考虑一下简单地调整快速排序算法，使之用于外部排序的情况。快速排序从处理整个记录组开始，第一次划分把索引从两端移到内部。这可以通过有效利用缓冲技术来实现。下一步就是处理每一个子记录组，接着处理子记录组的子记录组，以此类推。随着子记录组越来越小，处理很快变成对磁盘的随机访问。即使 I/O 操作可能很有效，平均情况下，快速排序处理每条记录仍然需要 $\log n$ 次。很快就会看到，还有比这种方法更好的方法。

1. 二路归并

进行外部排序的一个更好的方法源于归并排序。归并排序最简单的形式是对记录顺序地完成一系列扫描。在每一趟扫描中，归并的子列越来越大。而对于外部排序来说，基本上有两个相对独立的阶段组成。首先，按可用内存大小，将外存上含 n 个记录的文件分成若干长度为 l 的子文件或段(segment)，依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写入外存。通常称这些有序子文件为归并段或顺串；然后对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至整个有序文件为止。

二路归并的算法如下：

(1) 把原来的文件分成两个大小相等的顺串文件。

(2) 从每个顺串文件中取出一个块，读入输入缓冲区中。

(3) 从每个输入缓冲区中取出第 1 条记录，把它们按照排好的顺序写入一个顺串输出缓冲区中。

(4) 从每个输入缓冲区中取出第 2 条记录，把它们按照排好的顺序写入另一个顺串输出缓冲区中。

(5) 在两个顺串输出缓冲区之间交替输出，重复这些步骤直到结束。当一个输入块用完时，从相应的输入文件读出第二个块。当一个顺串输出缓冲区已满时，把它写回相应的输出文件。

(6) 使用原来的输出文件作为输入文件，重复(2)至(5)步。在第二趟扫描中，每个输入顺串文件的前两条记录已经排好了次序。这样一来，就可以把这两个顺串归并成一个长度为 4 个元素的顺串输出。

(7) 对顺串文件的每一趟扫描产生的顺串越来越大，直到最后只剩下一个顺串。

这个算法可以方便地使用前面讲述的双缓冲技术来实现。如果一个文件有 n 条记录，对这个文件进行简单的二路归并排序需要 $\lg n$ 趟扫描。这样，就需要对每条记录进行 $\lg n$ 次磁盘读/写。所以需要对二路归并算法进行优化。

虽然现在有很多外部排序算法的变种，但大多数变种都依据同样的原理。一般来说，外部排序的所有好算法都基于下面两步：

(1) 把文件分成大的初始顺串。

(2) 把所有顺串归并到一起，形成一个已排序的文件。

2. 置换选择排序

现在讨论怎样为一个磁盘文件创建尽可能大的初始顺串的问题。这里假定 RAM 大小是固定的。如果分配给数组的可用存储器大小是 M 条记录，那么就可以把输入文件分成长度为 M 的初始顺串。一种更好的方法是使用称为置换选择的算法。在平均情况下，这种算法可以创建长度为 $2M$ 条记录的顺串。置换选择实际上是堆排序算法的一个微小变种。虽然堆排序算法比快速排序算法慢，但相对于 I/O 时间来说是微乎其微的，所以影响并不大。

置换选择算法如下(假定主要处理在一个大小为 M 条记录的数组中完成)：

(1) 从磁盘中读出数据到数组中，设置 LAST=M - 1。

(2) 建立一个最小值堆(每个结点中记录的关键码值都小于其子孙结点中的关键码值)。

(3) 重复以下步骤，直到数组为空。

① 把具有最小关键码值的记录(根结点)送到输出缓冲区。

② 设 R 是输入缓冲区中的下一条记录。如果 R 的关键码值大于刚刚输出的关键码值，则把 R 放到根结点，否则使用数组中 LAST 位置的记录代替根结点，然后把 R 放到 LAST 位置，并设置 LAST=LAST - 1。

③ 筛出根结点，重新排列堆。

在置换算法的开始，几乎所有来自输入文件的值都比这个顺串的最新输出的关键码值大，因为这个顺串中的初始关键码值都很小。随着对顺串的处理，最新输出的关键码值变得越来越大，从而来自输入文件的新关键码值很可能太小，这些记录到了数组的底部。顺串的总长度预计是数组长度的两倍。

3. 多路归并

一般的外部排序算法在第二阶段归并第一阶段创建的顺串。如果使用简单的二路归

并，那么 R 个顺串对整个文件需要 $\lg n$ 趟扫描。尽管 R 应当远远小于记录总数(由于每个初始顺串都应当包含许多记录)，我们仍然希望进一步减少把顺串归并到一起需要的扫描趟数。二路归并并不能充分利用可用主存。由于归并是作用在两个顺串上的顺序过程，因此每个顺串一次只需要有一个块的记录在主存中。这样一来，置换选择算法的堆使用的大多数空间并没有在归并过程中使用。

如果一次归并多个顺串，就可以更好地利用这些空间，同时可以大大减少归并顺串需要的扫描趟数。

多路归并与二路归并类似。如果有 B 个顺串需要归并，从每个顺串中取出一个块放在主存中使用，那么 B 路归并算法仅仅查看 B 个值(每个输入顺串最前面的值)，并且选择最小的一个输出。把这个值从它的顺串中移出，然后重复这个过程。当任何顺串的当前块用完时，就从磁盘中读出这个顺串的下一块。

一般来说，建立大的初始顺串可以把运行时间减少到标准归并排序的四分之一，使用多路归并可以进一步把时间减半。

总之，一种好的外部排序算法会尽量做好以下几个方面：

- 尽量减少初始顺串。
- 在所有阶段尽可能把输入、处理和输出进行并行处理。
- 使用尽可能多的工作主存，以减少内外存交换次数，达到节约时间的目的，对外部排序而言，更快的 CPU 在运行时间方面影响不大。
- 使用多个外存(最好是随机存储设备)，使 I/O 处理具有更大的并行性，并允许顺序文件处理。

7.7 各种内排序方法的比较和选择

在前面几节中介绍了几种内排序的基本思想及算法，在实际应用时到底采用哪种排序方法呢？下面将通过几个标准来判断内排序方法的性能。

(1) 评价排序算法好坏的标准

评价排序算法好坏的标准主要有两条：

- 执行时间和所需的辅助空间
- 算法本身的复杂程度

(2) 排序算法的空间复杂度

若排序算法所需的辅助空间并不依赖于问题的规模 n，即辅助空间是 $O(1)$，则称之为就地排序(In-PlaceSou)。

非就地排序一般要求的辅助空间为 $O(n)$。

(3) 排序算法的时间开销

大多数排序算法的时间开销主要是关键字之间的比较和记录的移动。有的排序算法的

执行时间不仅依赖于问题的规模，还取决于输入实例中数据的状态。

下面从时间和空间两个方面比较各种排序方法的优缺点，以便在实际应用中选择合适的排序方法，如表 7-2 所示。

表 7-2　各种排序方法性能比较表

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
直接插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
冒泡	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔		$O(n^{1.25})$		$O(1)$	不稳定
快速	$O(n\lg n)$	$O(n\lg n)$	$O(n^2)$	$O(\lg n)$	不稳定
堆	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$	$O(1)$	不稳定
归并	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$	$O(n)$	稳定

思考和练习

(1) 试说明希尔排序的概念及基本思想。

(2) 试说明归并排序的概念及基本思想。

(3) 依次用下面 6 种排序算法对数组{44,77,55,99,66,33,22,88,77}进行排序，分别手动跟踪它们的排序过程。

- 冒泡排序
- 选择排序
- 插入排序
- 归并排序
- 快速排序
- 堆排序

(4) 堆排序与选择排序、插入排序有哪些相似之处？

(5) 写出下列排序法的程序，要求排序结果按降序排列。

- 插入排序
- 交换排序
- 选择排序
- 归并排序

第8章　查　　找

利用排序的方法实现查找满足不同条件的内容，是程序设计者必须掌握的。本章着重介绍几种查找的方法，程序设计者可以在实际的应用过程中灵活应用。

本章的学习目标：

- 线性表的查找方法；
- 二叉排序树的插入和生成及其相应的查找方法；
- B 树的插入和生成及其相应的查找方法；
- 散列函数的构造及其查找算法。

8.1 基 本 概 念

由于查找运算的使用频率很高，几乎在任何一个计算机系统软件和应用软件中都会涉及到，所以当问题涉及的数据量相当大时，查找方法的效率就显得格外重要。在一些实时查询系统中尤其如此。因此，本章将系统地讨论各种查找方法，并通过对它们的效率分析来比较各种查找方法的优劣。

1. 查找表和查找

一般地，假定被查找的对象是由一组结点组成的表(Table)或文件，而每个结点则由若干个数据项组成。并假设每个结点都有一个能惟一标识该结点的关键字。

查找(Searching)的概念是：给定一个值 K，在含有 n 个结点的表中找出关键字等于给定值 K 的结点。若找到，则查找成功，返回该结点的信息或该结点在表中的位置；否则查找失败，返回相关的指示信息。

2. 查找表的数据结构表示

(1) 动态查找表和静态查找表

若在查找的同时对表做修改操作(如插入和删除)，则相应的表称之为动态查找表。否则称之为静态查找表。

(2) 内查找和外查找

和排序类似，查找也有内查找和外查找之分。若整个查找过程都在内存进行，则称之为内查找；反之，若查找过程中需要访问外存，则称之为外查找。

3. 平均查找长度

查找运算的主要操作是关键字的比较，所以通常把查找过程中对关键字需要执行的平均比较次数(也称为平均查找长度)作为衡量一个查找算法效率优劣的标准。

平均查找长度(Average Search Length，ASL)定义为：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中，n 是结点的个数。p_i 是查找第 i 个结点的概率。若不特别声明，认为每个结点的查找概率相等，即 $p_1=p_2=\cdots=p_n=1/n$。c_i 是找到第 i 个结点所需进行的比较次数。

8.2　线性表查找

8.2.1　顺序查找

1. 基本思想

在表的组织方式中，线性表是最简单的一种。顺序查找是一种最简单的查找方法。其基本思想是：从表的一端开始，顺序扫描线性表，依次扫描到的结点关键字和给定的 K 值相比较，若当前扫描到的结点关键字与 K 相等，则查找成功；若扫描结束后，仍未找到关键字等于 K 的结点，则查找失败。

2. 顺序查找的存储结构要求

顺序查找方法既适用于线性表的顺序存储结构，也适用于线性表的链式存储结构(使用单链表作存储结构时，扫描必须从第一个结点开始)。

向量存储的线性表查找包括对有序表和无序表的查找。

3. 具体算法

按照顺序查找的基本思想，其具体算法如下：

```
// ==========Program Descrition==========
// 程序名称：lsearch.java
// 程序目的：设计一个线性查找程序。
// ======================================
import java.util.*;
import java.io.*;
public class lsearch
{       public static int[] Data =
          { 12,76,29,22,15,
```

```
            62,29,58,35,67,
            58,33,28,89,90,
            28,64,48,20,77};
    public static int Counter=1;//查找次数计数变量
public static void main (String args[])
    {           int KeyValue=0;
                    //输入欲查找值
System.out.print("Please enter your key value:");
                //读入输入数值
InputStreamReader is=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(is);
    StringTokenizer st;
      try{
            String myline=br.readLine();
            st=new StringTokenizer(myline);
            KeyValue=Integer.parseInt(st.nextToken());//取得输入值
          }
      catch(IOException ioe)
          {
            System.out.print("IO error:"+ioe);
          }              //调用线性查找
      if (Linear_Search((int) KeyValue))
      {       System.out.println("");     //输出查找次数
              System.out.println("Search Time="+(int) Counter);
        }
      else
      {         System.out.println("");       //输出没有找到数据
            System.out.print("No Found!! ");
      }
 }
//-----------------------------------------------
//顺序查找
//-----------------------------------------------
    public static boolean Linear_Search(int key)
{           int i;//数组下标变量
      for (i=0;i<20;i++)
      {       System.out.print("["+(int) Data[i]+"]");     //输出数据
              if((int) key == (int) Data[i])                //查到数据时
                    return true;//返回 true
               Counter++;       //计数器递增
      }
      return false;           //返回 false
}}
```

4. 算法分析

成功时的顺序查找的平均查找长度为：

$$ASL_{sq}=\sum_{i=1}^{n}p_ic_i=\sum_{i=1}^{n}(n-i+1)=np_1+(n-1)p_2+\ldots+2p_{n-1}+p_n$$

在等概率情况下，$p_i=1/n(1\leqslant i\leqslant n)$，故成功的平均查找长度为：

$$(n+\cdots+2+1)/n=(n+1)/2$$

即查找成功时的平均比较次数约为表长的一半。

若 K 值不在表中，则须进行 $n+1$ 次比较之后才能确定查找失败。

顺序查找的优点是算法简单，且对表的结构无任何要求，无论是用向量还是用链表来存放结点，也无论结点之间是否按关键字有序，它都同样适用。其缺点是查找效率低，因此，当 n 较大时不宜采用顺序查找。

8.2.2　二分查找

1. 基本思想

二分查找又称折半查找，它是一种效率较高的查找方法。其要求：线性表是有序表，即表中结点按关键字有序，并且要用向量作为表的存储结构。不妨设有序表是递增有序的。

二分查找的基本思想是：先确定待查找记录所在的范围(区间)，然后逐步缩小范围直到找到或找不到该记录为止。

(1) 首先确定该区间的中点位置：

$$\text{mid}=(\text{low}+\text{high})/2$$

(2) 然后将待查的 K 值与 R[mid].key 比较。若相等，则查找成功并返回此位置，否则须确定新的查找区间，继续二分查找，具体方法如下：

① 若 R[mid].key>K，则由表的有序性可知 R[mid…n].keys 均大于 K，因此若表中存在关键字等于 K 的结点，则该结点必定是在位置 mid 左边的子表 R[1…mid - 1]中，故新的查找区间是左子表 R[1…mid - 1]。

② 类似地，若 R[mid].key<K，则要查找的 K 必在 mid 的右子表 R[mid+1…n]中，即新的查找区间是右子表 R[mid+1…n]。下一次查找是针对新的查找区间进行的。

因此，从初始的查找区间 R[1…n]开始，每经过一次与当前查找区间的中点位置上的结点关键字的比较，就可确定查找是否成功，不成功则当前的查找区间就缩小一半。这一过程重复直至找到关键字为 K 的结点，或者直至当前的查找区间为空(即查找失败)时为止。

2. 具体算法

按照二分查找的基本思想，可以采用非递归方式设计二分查找法，也可以采用递归方式实现。下面分别加以描述。

(1) 非递归方式设计二分查找法

```
//=====================Program Discription=====================
//程序名称：bsearch.java
//程序目的：运用非递归方式设计二分查找法的程序。
//==============================================================
import java.util.*;
import java.io.*;
public class bsearch
{       public static int Max=20;
      public static int[] Data=
              { 12,16,19,22,25,
                32,39,48,55,57,
                58,63,68,69,70,
                78,84,88,90,97};//数据数组
      public static int Counter=1;//查找次数计数变量
public static void main(String args[])
    {       int KeyValue=0;
      System.out.print("Please enter your key value:");   //输入欲查找值
    InputStreamReader is=new InputStreamReader(System.in);   //读入输入数值
      BufferedReader br=new BufferedReader(is);
         StringTokenizer st;
           try{   String myline=br.readLine();
                  st=new StringTokenizer(myline);
                  KeyValue=Integer.parseInt(st.nextToken());//取得输入值
                }
           catch(IOException ioe)
                {       System.out.print("IO error:"+ioe);
                }
      if (BinarySearch((int) KeyValue))       //调用二分查找
            { System.out.println("");
            System.out.println("Search Time="+(int) Counter);     //输出查找次数
            }
      else
      {     System.out.println("");
            System.out.println("No Found!!");   //输出没有找到数据
      }}
//--------------------------------------------------------
//二分查找法
```

```
//-------------------------------------------------------------
    public static boolean BinarySearch (int KeyValue)
{     int Left;     //左边界变量
      int Right;    //右边界变量
      int Middle; //中位数变量
        Left=0;
      Right=Max-1;
      while (Left<=Right)
      {     Middle=(Left+Right)/2;
            if(KeyValue<Data[Middle])  //欲查找值较小
                  Right=Middle-1;          //查找前半段
            else if (KeyValue>Data[Middle])   //欲查找值较大
                 Left=Middle+1;//查找后半段
            else if (KeyValue==Data[Middle])  //查找到数据
            {     System.out.println("Data["+Middle+"]="+Data[Middle]);
                  return true;
            }
            Counter++;
      }
      return false;
}}
```

(2) 递归方式设计二分查找法

```
//=====================Program Discription=============================
//程序名称：rbsearch.java
//程序目的：运用递归方式设计二分查找法的程序。
//=====================================================================
import java.util.*;
import java.io.*;
public class rbsearch
{       public static int Max=20;
      public static int[] Data=
                  { 12,16,19,22,25,
                    32,39,48,55,57,
                    58,63,68,69,70,
                    78,84,88,90,97};//数据数组
    public static int Counter=1;//查找次数计数变量
public static void main(String args[])
    {         int KeyValue=0;
      System.out.print("Please enter your key value:");   //输入欲查找值
      InputStreamReader is=new InputStreamReader(System.in); //读入输入数值
      BufferedReader br=new BufferedReader(is);
         StringTokenizer st;
```

```
try{   String myline=br.readLine();
       st=new StringTokenizer(myline);
       KeyValue=Integer.parseInt(st.nextToken());//取得输入值
    }
catch(IOException ioe)
    { System.out.print("IO error:"+ioe);
    }
       if (BinarySearch((int) KeyValue))          //调用二分查找
            {      System.out.println("");
            System.out.println("Search Time="+(int) Counter);   //输出查找次数
            }
       else
       { System.out.println("");
        System.out.println("No Found!!");       //输出没有找到数据
       }}
//-----------------------------------------------------
//递归二分查找法
//-----------------------------------------------------
    public static boolean RBinarySearch (int Left,int Right,int KeyValue)
 {     int Middle;        //中位数变量
  Counter++;
   if (Left>Right)
         return false;
   else
   {     Middle=(Left+Right)/2;
          if(KeyValue<Data[Middle])          //欲查找值较小
         return RBinarySearch(Left,Middle-1,KeyValue);    //查找前半段
         else if (KeyValue>Data[Middle])     //欲查找值较大
        return RBinarySearch(Middle+1,Right,KeyValue);   //查找后半段
         else if (KeyValue==Data[Middle])    //查找到数据
   {       System.out.println("Data["+Middle+"]="+Data[Middle]);
            return true;
         }}
   return false;
  }}
```

3. 算法分析

二分查找过程可用二叉树来描述：把当前查找区间的中间位置上的结点作为根，把左子表和右子表中的结点分别作为根的左子树和右子树。由此得到的二叉树称为描述二分查找的判定树(Decision Tree)或比较树(Comparison Tree)。

设内部结点的总数为 $n=2^h-1$，则判定树是深度为 $h=\lg(n+1)$的满二叉树(深度 h 不计外

部结点)。树中第 k 层上的结点个数为 2^{k-1}，查找它们所需的比较次数是 k。因此在等概率假设下，二分查找成功时的平均查找长度为：

$$ASL \approx \lg(n+1)-1$$

二分查找在查找失败时所需比较的关键字个数不超过判定树的深度，在最坏情况下查找成功的比较次数也不超过判定树的深度。即为：

$$\left\lfloor \log_2^n \right\rfloor +1$$

二分查找的最坏性能和平均性能相当接近。

虽然二分查找的效率高，但是要将表按关键字排序。而排序本身是一种很费时的运算。既使采用高效率的排序方法也要花费 O($n$lg$n$)的时间。

二分查找只适用顺序存储结构。为保持表的有序性，在顺序结构里插入和删除都必须移动大量的结点。因此，二分查找特别适用于那种一经建立就很少改动而又经常需要查找的线性表。

对那些查找少而又经常需要改动的线性表，可采用链表作存储结构，进行顺序查找。链表上无法实现二分查找。

8.2.3　分块查找

分块查找又称索引顺序查找。它是把顺序查找和二分查找相结合的一种查找方法，即把线性表分成若干块，块和块之间有序，但每一块内的结点可以无序。分块查找的基本思想是：确定被查找的结点所在的块(采用二分查找法)后，对该块中的结点采用顺序查找。

分块查找介于顺序和二分查找之间，其优点是：在表中插入或删除一个记录时，只要找到该记录所属的块，就在该块内进行插入和删除运算。分块查找的主要代价是增加一个辅助数组的存储空间和将初始表分块排序的运算。

8.3　二叉排序树

当用线性表作为表的组织形式时，可以用 3 种查找法。其中以二分查找效率最高。但由于二分查找要求表中结点按关键字有序，且不能用链表作存储结构，因此，当表的插入或删除操作频繁时，为维护表的有序性，势必要移动表中很多结点。这种由移动结点引起的额外时间开销，就会抵消二分查找的优点。也就是说，二分查找只适用于静态查找表。若要对动态查找表进行高效率的查找，最好使用二叉排序树。

1. 二叉排序树的基本概念

二叉排序树(Binary Sort Tree)又称二叉查找(搜索)树或二叉分类树，它是一种特殊的二叉树：或者为空或者满足下面的条件。

(1) 每个结点左子树上的所有结点的关键字，值均小于该结点的关键字值；

(2) 每个结点右子树上的所有结点的关键字，值均大于或等于该结点的关键字值；

(3) 左、右子树本身又各是一棵二叉排序树。

上述性质简称二叉排序树性质(BST 性质)，故二叉排序树实际上是满足 BST 性质的二叉树。

2. 二叉排序树的特点

由 BST 性质可得：

(1) 二叉排序树中任一结点 x，其左(右)子树中任一结点 y(若存在)的关键字必小(大)于 x 的关键字。

(2) 二叉排序树中，各结点关键字是惟一的。

需要注意的是在实际应用中，不能保证被查找的数据集中各元素的关键字互不相同，所以可将二叉排序树定义中 BST 性质(1)里的“小于”改为“大于等于”，或将 BST 性质(2)里的“大于”改为“小于等于”，甚至可同时修改这两个性质。

(3) 按中序遍历该树所得到的中序序列是一个递增有序序列。

3. 二叉排序树的插入和生成

假若给定一个元素序列，可以利用上述算法创建一棵二叉排序树。首先，将二叉树序树初始化为一棵空树，然后逐个读入元素，每读入一个元素，就建立一个新的结点插入到当前已生成的二叉排序树中，即调用上述二叉排序树的插入算法将新结点插入。生成二叉排序树的算法如下。

已知一个关键字值为 key 的结点 s，若将其插入到二叉排序树中，只要保证插入后仍符合二叉排序树的定义即可。插入可以用下面的方法进行：

(1) 若二叉树序树是空树，则 key 成为二叉树序树的根；

(2) 若二叉树序树非空，则将 key 与二叉树序树的根进行比较，如果 key 的值等于根结点的值，则停止插入，如果 key 的值小于根结点的值，则将 key 插入左子树，如果 key 的值大于根结点的值，则将 key 插入右子树。

具体算法如下：

```
public void Create(int Data)
        {           int i;              //循环计数变量
                    int Level=0;        //树的层数
                    int Position=0;
                for(i=0;TreeData[i] !=0;i++);
                TreeData[i]=Data;
```

```
while (true)                                    //寻找结点位置
{   if(Data>TreeData[Level])                    //判断是左子树还是右子树
        {   if(RightNode[Level] != -1)          //判断右子树是否有下一层
                    Level=RightNode[Level];
              else
              { Position= -1;                   //设定为右子树
                    break;
              } }
          else
          { if(LeftNode[Level] != -1)           //判断左子树是否有下一层
                  Level=LeftNode[Level];
                else
                {   Position= 1;                //设定为右子树
                    break;
                } } }
        if(Position==1)                         //判定结点的左右连接
         LeftNode[Level]=i;                     //左连接
        else
         RightNode[Level]=i;                    //右连接
}
```

4. 二叉排序树的算法

实际上二叉排序树的算法就是一个动态的查找过程。其过程可以简单描述为：在查找某一个值 K 时，首先令 K 与二叉树的根结点的值 R_a 进行比较；如果 $K<R_a$，则在二叉树的左子树上查找 K；如果 $K\geqslant R_a$，则在二叉树的右子树上查找 K；如果二叉排序树中没有值为 K 的结点，则将值为 K 的结点按照二叉排序树构造的规则插入到该二叉树中。

其对应算法主要分为两步，建立二叉树和对结点的查找。具体算法如下：

```
//=========================Program Discription=================================
//程序名称：btreesearch.java
//程序目的：设计二叉排序树的算法及程序。
//=============================================================================
import java.util.*;
import java.io.*;
public class btreesearch
{       public static int Max=10;
      public static int[] Data=
              { 15,2,13,6,17,
                25,37,7,3,18};       //数据数组
      public static int Counter=1;
 public static void main(String args[])
      {     int i;                          //循环变量
```

```
        BNTreeArray BNTree=new BNTreeArray();      //声明二叉数组
            //BNTree.Create(Data[0])=Data[0];
for (i=0;i<Max;i++)
        BNTree.Create(Data[i]);       //建立二叉排序树
    System.out.print("Please enter your key value:");
    InputStreamReader is=new InputStreamReader(System.in);   //读入数值
BufferedReader br=new BufferedReader(is);
            StringTokenizer st;
              try{   String myline=br.readLine();
                     st=new StringTokenizer(myline);
                     KeyValue=Integer.parseInt(st.nextToken());//取得输入值
                   }
              catch(IOException ioe)
                   { System.out.print("IO error:"+ioe);
                   }
              if (BNTree.BinarySearch(KeyValue)>0)   //调用二叉树查找法
System.out.println("Search Time="+BNTree.BinarySearch(KeyValue));   //输出查找次数
        Else
              System.out.println("No Found!!");     //输出没有找到数据
 }}
class BNTreeArray
{           public static int MaxSize=20;
          public static int[] TreeData=new int[MaxSize];
          public static int[] RightNode=new int[MaxSize];
          public static int[] LeftNode=new int[MaxSize];

     public BNTreeArray()
     {      int i;
            for (i=0;i<MaxSize ;i++ )
            {      TreeData[i]=0;
                RightNode[i]= -1;
                   LeftNode[i]=1;
            }  }
//-------------------------------------------------------
//建立二叉树
//-------------------------------------------------------
     public void Create(int Data)
       {            int i;              //循环计数变量
                    int Level=0;        //树的层数
                    int Position=0;
              for(i=0;TreeData[i] !=0;i++);
              TreeData[i]=Data;
              while (true)                                  //寻找结点位置
```

```
{   if(Data>TreeData[Level])                    //判断是左子树还是右子树
        {   if(RightNode[Level] != -1)          //判断右子树是否有下一层
                        Level=RightNode[Level];
            else
            { Position= -1;                     //设定为右子树
                  break;
            } }
          else
          { if(LeftNode[Level] != -1)     //判断左子树是否有下一层
                  Level=LeftNode[Level];
                else
                {   Position= 1;          //设定为左子树
                    break;
                } } }
          if(Position==1)                           //判定结点的左右连接
           LeftNode[Level]=i;                       //左连接
          else
           RightNode[Level]=i;                      //右连接
      }
//-----------------------------------------------------
//二叉树查找法
//-----------------------------------------------------
    public static int BinarySearch(int KeyValue)
{     int Pointer;       //现结点位置
      int Counter;     //查找次数
      Pointer=0;
      Counter=0;
      while (Pointer!= -1)
      {Counter++;
          if (TreeData[Pointer]==KeyValue)     //找到了欲查找的结点
              return Counter;                   //返回查找次数
          else if (TreeData[Pointer]>KeyValue)
              Pointer=LeftNode[Pointer];       //找左子树
            else
                Pointer=RightNode[Pointer];     //找右子树
      }
      return 0;                                 //该结点不在此二叉树中
}}
```

5. 算法分析

在二叉排序树上进行查找时，若查找成功，则是从根结点出发走了一条从根到待查结点的路径。若查找不成功，则是从根结点出发走了一条从根到某个叶子的路径。

(1) 二叉排序树查找成功的平均查找长度

在等概率假设下，二叉排序树查找成功的平均查找长度为：

$$ASL = \sum_{1}^{n} p_i c_i$$

(2) 在二叉排序树上进行查找时的平均查找长度和二叉树的形态有关

二分查找法查找长度为 n 的有序表，其判定树是惟一的。含有 n 个结点的二叉排序树却不惟一。对于含有同样一组结点的表，由于结点插入的先后次序不同，所构成的二叉排序树的形态和深度也可能不同。

- 在最坏情况下，二叉排序树是通过把一个有序表的 n 个结点依次插入而生成的，此时所得的二叉排序树蜕化为深度为 n 的单支树，它的平均查找长度和单链表上的顺序查找相同，亦是$(n+1)/2$。
- 在最好情况下，二叉排序树在生成的过程中，树的形态比较匀称，最终得到的是一棵形态与二分查找的判定树相似的二叉排序树，此时它的平均查找长度大约是 $\lg n$。
- 插入、删除和查找算法的时间复杂度均为 $O(\lg n)$。

(3) 二叉排序树和二分查找的比较

就平均时间性能而言，二叉排序树上的查找和二分查找差不多。

就维护表的有序性而言，二叉排序树无须移动结点，只需修改相关值即可完成插入和删除操作，且其平均的执行时间均为 $O(\lg n)$，因此更有效。二分查找所涉及的有序表是一个向量，若有插入和删除结点的操作，则维护表的有序性所花的代价是 $O(n)$。当有序表是静态查找表时，宜用向量作为其存储结构，而采用二分查找实现其查找操作；若有序表为动态查找表，则应选择二叉排序树作为其存储结构。

(4) 平衡二叉树

为了保证二叉排序树的高度为 $\lg n$，从而保证二叉排序树上实现的插入、删除和查找等基本操作的平均时间为 $O(\lg n)$，在往树中插入或删除结点时，要调整树的形态来保持树的“平衡”。使之既保持 BST 性质不变又保证树的高度在任何情况下均为 $\lg n$，从而确保树上的基本操作在最坏情况下的时间均为 $O(\lg n)$。需要说明的是：

- 平衡二叉树(Balanced Binary Tree)是指树中任一结点的左右子树的高度大致相同。
- 任一结点左右子树的高度均相同(如满二叉树)，则二叉树是完全平衡的。通常，只要二叉树的高度为 $\lg n$，就可看作是平衡的。
- 平衡的二叉排序树指满足 BST 性质的平衡二叉树。

8.4　B 树

对于计算机系统来说，CPU(Centre Processing Unit)处理数据的速度是微秒级或纳秒级，

比磁盘或磁带快百万倍，甚至更快，而数据库程序中大部分信息都存储在磁盘或磁带上，这样辅存上的信息处理会显著地降低程序的执行速度。因此对基于磁盘或磁带的数据进行有效的检索时，选择正确的数据结构以大大减少访问这些辅存的时间代价就显得非常重要，而 B 树就是这样的方法。

1. m 路查找树

与二叉排序树类似，可以定义一种“*m* 叉排序树”，通常称为 *m* 路查找树。

一棵 *m* 路查找树，或者是一棵空树，或者是满足如下性质的树：

- 结点最多有 *m* 个子女，*m* - 1 个关键字。
- 每个结点中的值是按升序排列的。
- 前 *i* 个子女的值比第 *i* 个值小。
- 后 *m* - *i* 个子女的值比第 *i* 个值大。

如图 8 - 1 所示是一棵 3 路查找树。

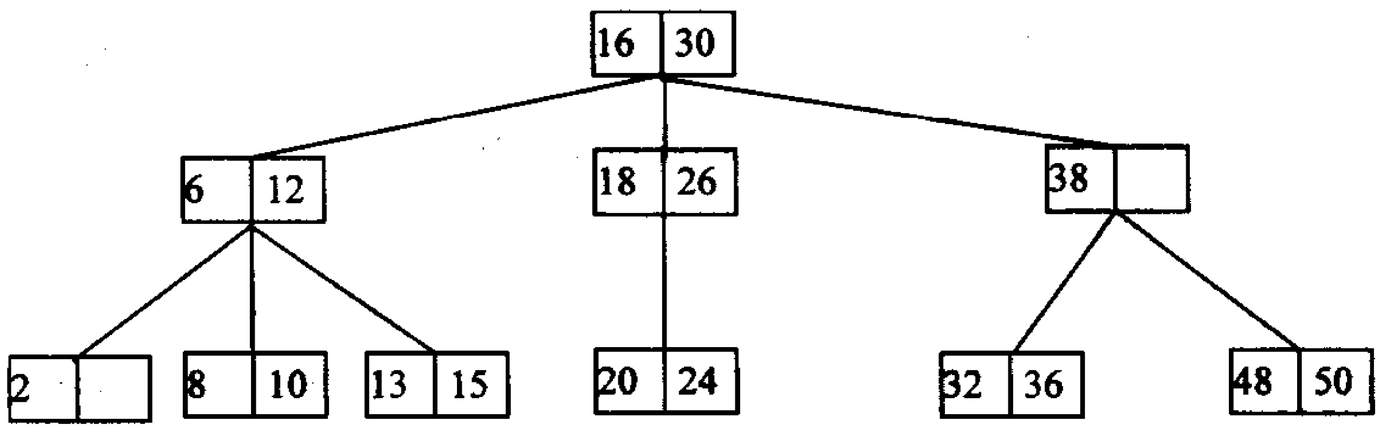


图 8 - 1　3 路查找树

为了更好地学习 B 树，下面首先讨论 3 路查找树的查找和插入。

(1) 平衡 3 路查找树及其查找

由 m 路查找树的性质我们很容易得出 3 路查找树的特征：

- 一个结点包含一个或两个关键码。
- 每个内部结点最多有 3 个子女。
- 所有叶结点都在树的同一层，因此树的高度总是平衡的。

平衡 3 路查找树的 Java 类说明如下：

```
class TTNode {
   private Elem lkey;
   private Elem rkey;        //结点的左右值
   private int numKeys;      //结点存储值的数量
   private TTNode left;
private TTNode center;
private TTNode right;        //结点的左、中、右子树
public TTNode(){center=left=right=null;numkeys=0;}
public TTNode(Elem l,Elem r,TTNode p1,TTNode p2,TTNode p3){
   lkey=l;rkey=r;
   if (r==null)
```

```
        numkeys=1;
      else
        numkeys=2;
      left=p1;
center=p2;
right=p3
}
      //结点内部增加值和子树
public void addKey(Elem val, TTNode child) {
Assert.notFalse(numKeys==1,"Illegal addkey");
numKeys=2
if (val.key()<lkey.key()) {
rkey=lkey;
lkey=val;
right=center;
center=child;}
}
public int numKeys() {return numKeys;}
public TTNode lchild() {return left;}
public TTNode rchild() {return right;}
public TTNode cchild() {return center;}
public Ttnode setCenter(TTNode val) {return center=val;}
public Elem lkey() {return lkey;}
public Elem rkey() {return rkey;}
public boolean isLeaf() {return left==null;}
      //修改结点的左值
public void setLkey(Elem val) {
   Assert.notFalse(val!=null,"Can't nullify left value");
   Lkey=val;
}
      //修改结点的右值
public void setRkey(Elem val) {
   Assert.notFalse(val!=0,"Can't set right key of empty node");
   if ((val==null)&&(numKeys==2)) {
      {right=null; numKeys=1;}
rkey=val;
}
}
```

3 路查找树的查找过程与二叉排序树的查找过程类似。查找从根结点开始，如果根结点不包含查找关键码，就在可能包含关键码的子结点中继续查找。具体算法如下：

```
private Elem TTsearch(Ttnode root, int val) {
   if (root==null) return null;        //根为空，值未找到
```

```
    if (val==root.lkey().key())
return root.lkey();
    if ((root.numKeys().key()==2) && (val==root.rkey().key()))
       return root.rkey();                                    //根不空，值找到
    if (val<root.lkey().key())                                //查找左子树
       return TTsearch(root.lchild(),val);
    else   if (root.numKeys()==1)                             //查找中子树
          return TTsearch(root.cchild(),val);
       else if (val<root.rkey().key())                        //查找中子树
             return TTsearch(root.cchild(),val);
          else                                                //查找右子树
             return TTsearch(root.rchild(),val);

}
```

(2) 3 路查找树的插入和生成

向一个 3 路查找树插入记录类似于向一个 BST 插入记录，新记录也放到相应的叶结点中。与 BST 插入记录不同，3 路查找树并不创建新的子女结点放置插入的记录。如果关键码在树中，第一步是找到将会包含这个关键码的叶结点。如果这个叶结点只包含一个值，则不需要对树进行修改就可以把新关键码添加到这个结点中。如果这个叶结点已经包含两个关键码，则需要创建更多的空间。这时必须从空闲存储区中创建一个新结点，将原结点中的两个关键码值和新关键码值中最小的一个放入原结点，最大的一个放入新结点，而中间的关键码值与指向新结点的指针传回到父结点。这称为一次提升。如果父结点只包含一个关键码值，则直接将传回的中间关键码值插入；如果父结点已包含两个关键码值，则就重复提升过程。具体 Java 实现如下：

```
private Object[] TTcreate(TTNode rt, Elem val) {
   if (rt==null) {                                  //根结点为空
rt=new TTNode(val,null,null,,null,null);
Object[] temp={rt}
}
if (rt.isLeaf())                                    //在叶结点插入
   if (rt.numKeys()==1)
      {rt.addKey(val,null); return null;}
   else return splitnode(rt,val,null);
                        //在结点内部
Object[] retval;
if (val.key()<rt.lkey().key())
   retval=Ttcreate(rt.lchild().val);
else if ((rt.numKeys()==1) || (val.key()<rt.rkey().key()))
         retval=Ttcreate(rt.cchild(),val);
     else retval=Ttcreate(rt.rchild(),val);
if (retval==null) return null;
```

```
                    //分裂子结点
if (rt.numkeys()==1) {
rt.addKey((Elem)retval[0],(TTNode)retval[1]);
return null;
}
                //同时分裂当前结点
return splitnode(rt,(Elem)retval[0],(TTNode)retval[1]);
}
                //分裂一个结点
Object[] splitnode(TTNode rt, Elem val, TTNode child) {
Object[] temp= new Object[2];
if (val.key()>rt.rkey().key()) {     //比结点右值大
   temp[0]=rt.rkey();                  //提升右值
   TTNode hold=rt.rchild;
   rt.setRkey(null);
   temp[1]=new TTNode(val,null,hold,child,null);
}
else if (val.key()>rt.lkey().key()) { //比结点左值大
   temp[0]=val;                        //提升待插入值
   temp[1]=new TTNode(rt.rkey(),null,child,rt.rchild(),null);
   rt.setRkey(null);
}
else {                                 //比结点左值小
   temp[0]=rt.lkey();                  //提升左值
   temp[1]=new TTNode(rt.rkey(),null,rt.cchild,rt.rchild(),null);
   rt.setCenter(child);
   rt.setLkey(val);
rt.setRkey(null);
}
          return temp;      //返回提升的关键码值和它的子树
}
```

2. B 树及其查找

(1) B 树的基本概念

一棵 B 树是一棵平衡的 m 路查找树，B 树的研究通常归功于 R.Bayer 和 E.McCreight，他们在 1972 年的论文中描述了 B 树。到 1979 年，B 树几乎已经替代了除散列方法以外的所有大型文件访问方法。B 树的一个重要属性是每个结点的大小可以和磁盘或磁带中的一个块的大小相同，所以 B 树涉及了当实现基于磁盘或磁带的查找树时遇到的所有问题。

一棵 B 树或者是空树，或者是满足如下性质的树：

- 树中每个结点最多有 m 棵子树；
- 根结点至少有两棵子树；
- 除根结点之外的所有非叶结点至少有$\lceil m/2 \rceil$棵子树；

- 所有叶结点出现在同一层上。

B 树是平衡二叉排序树或 3 路查找树的一种推广，从另一方面来说，平衡二叉排序树是一个 2 阶 B 树，平衡 3 路查找树是一个 3 阶 B 树。

(2) B 树的查找

B 树的查找是平衡 3 路查找树的查找算法的一种推广，它是一个交替的两步过程，主要的区别是在当前结点查找时使用二分查找算法。

给定关键码值 k，为了在 m 阶 B 树中查找对应记录，具体算法如下：

- 如果树为空，返回 null。
- 令 x 为根结点。
- 重复步骤(4)至(6)，直至 x 为叶结点。
- 在结点 x 使用二分查找算法查找关键码值 K_i，使得 $K_{i-1}<k\leqslant K_i$ ($K_0=-\infty$，$K_m=+\infty$)。
- 如果 K_i =k，从磁盘读取相应记录并返回之，否则执行(6)。
- p_i 为 x 结点重复(4)(5)
- 查找失败返回 null。

(3) B 树的插入和生成

同样，B 树的插入和生成也是平衡 3 路查找树的插入和生成算法的一种推广。

给定关键码值 k，为了在 m 阶 B 树中插入对应记录，具体算法如下：

- 如果树为空，创建一个带有两个空叶结点的树根，将 k 插入到根结点上，并返回 True(表示插入成功)。
- 令 x 为根结点。
- 重复步骤(4)至(6)，直至 x 为叶结点。
- 在结点 x 使用二分查找算法查找关键码值 K_i，使得 $K_{i-1}<k\leqslant K_i$ ($K_0=-\infty$，$K_m=+\infty$)。
- 如果 $K_i=k$，返回 False(表示插入失败，因为已经存在一个惟一的 k 值)。
- 令 x 为子树 S_i 的根结点。
- 将记录存入磁盘或磁带。
- 在结点 x 的关键码值 K_{i-1} 和 K_i 之间插入 k(及记录的磁盘地址)。
- 在结点 x 加入一个空叶结点。
- 如果当前树高 degree(x)=m，重复步骤(11)至(13)直到 degree(x)<m。
- 令 K_j 为结点 x 的中间关键码值。
- 从 x 中将 K_j 移出将使原来的关键码值分为两部分，令 u 和 v 分别为左、右部分。
- 如果结点 x 为根结点，创建一个包含 K_j 并且以 u 和 v 为左右子树的新根结点。
- 否则，将 K_j 插入到 x 的父结点中并维护父结点同 u、v 的连接。
- 返回 True。

从上述 B 树的构造过程可得出以下结论：

- 由于 B 树是“从叶往根”长，而根对每个分支是公用的，所以不论根长到多“深”，各分支的长度同步增长，因而各分支是“平衡”的。

- 生长的几种情况：
- 最底层某个结点增大，分支数不变，且各分支深度也不变。
- 从最下层开始，发生单次或连续分裂，但根结点未分裂，此时分支数增 1(最下层结点增 1)，但原分支深度不变，新分支深度与原分支相同。
- 从最下层开始，连续分裂，根结点也发生分裂，产生一个新的根结点，此时分支数仍增 1(最下层结点增 1)，但新、旧分支均为原分支长度加 1。

根据 B 树的定义，必须保证 B 树半满，因此可能有 50%的空间被浪费。如果经常发生这种情况，则必须重新考虑这个定义或者对 B 树增加其他的限制。一般情况下，B 树大约 69%满，不太可能全部放满，所以最好再有一些附加约束。

3. B 树家族

(1) B*树

因为 B 树中的每个结点都代表了一个辅存块，访问一个结点就意味着一次辅存的存取，这相对于主存中结点的访问来说，代价太大。所以，创建的结点越少越好。

一个 B*树是 B 树的衍生物，它由 Donald Knuth 提出，由 Douglas Comer 命名。在一个 B*树中，除了根的所有结点都需要三分之二满，而不像 B 树中的半满。更精确地说，所有 m 阶的 B 树的非根结点中的关键码数是$(2m-1)/3-1<k\leqslant m-1$。结点分裂的频率通过延迟结点分裂而降低，当时机到来时，也是将两个结点分成三个结点，而不是一个分成两个。B*树的平均利用率大约为 81%。

(2) B^+树

由于一棵 B 树的结点代表了一个辅存块，从一个结点到另一个结点需要费时的块切换。所以，最好是尽可能地少访问结点。如果请求以升序访问 B 树中的结点会怎么样呢？可以用易于实现的中序遍历，但是对非末端结点，一次只能显示一个键，然后就要访问其他的块了。因此，我们更希望能增强 B 树，以便能以比中序遍历快的方式顺序访问数据。B^+树就提供了一个解决方案。

B^+树和 BST 及前面介绍的 3 路查找树最显著的差异是 B^+树只在叶结点存储记录。内部结点存储关键码值，但这些关键码值只是占据位置，用于引导查找，这意味着内部结点在结构上与叶结点有着显著的差异。内部结点存储关键码来引导查找，把每个关键码与指向子女结点的指针关联起来，叶结点存储实际记录。

8.5 散 列 技 术

1. 散列表的概念

(1) 散列(HASH)表的定义

若一个结点在表中的位置和该结点的关键字之间不存在确定的关系，则在表中查找某

结点时必然要进行关键字的比较，否则不然。

通常情况下，真正要存储的关键字数目比可能的关键字总数少得多。即实际发生的关键字集合往往是所有可能取值的关键字集合的一个很小的子集。若要使用直接寻址，就必须为每个可能的关键字在向量空间中保留一个位置。因此，最保守的估计是向量空间的大小为关键字数目足以满足实际的需求。

将所有可能出现的关键字集合记为 U，简称全集，实际发生即实际存储的关键字集合记为 K。

虽然直接寻址技术可在 $O(1)$时间内存取表中的任一结点，但它有一个致命的缺陷：当 U 很大时，要存储一个规模为$|U|$的表 T 是不实际的，有时甚至是不可能的，这是因为$|U|$可能远远大于主存的容量。既使$|U|$不太大，但只要$|K|$比$|U|$小得多时，T 的大部分空间亦被浪费，此时 T 是一个稀疏表。因此，必须将 T 的空间压缩至 $O(|K|)$的规模，但这样直接寻址技术就可能不再适用了。我们必须在关键字和表地址之间建立一个对应关系 h，它将全集 U 映射到表 $T[0\cdots m-1]$的下标集上(这里 $m=O(|K|)$。

$$h:\ U->\{0,\ 1,\ 2,\ \cdots,\ m-1\}$$

通常称 h 为散列函数，T 为散列表。对于任意的关键字 $Ki\in U$，包含此关键字的结点被存储在表 T 的 $h(Ki)$位置上，也就是说散列函数的自变量是 U 中的关键字，函数值为相应结点的存储地址(亦称散列值或散列地址)。将结点按其关键字的散列地址存储到散列表中的过程称为散列。

(2) 散列表的冲突现象

① 冲突

两个不同的关键字，由于散列函数值相同，因而被映射到同一表位置上。该现象称为冲突(Collision)或碰撞。发生冲突的两个关键字称为该散列函数的同义词(Synonym)。

② 安全避免冲突的条件

最理想的解决冲突的方法是安全避免冲突。要做到这一点必须满足两个条件：其一是$|U|\leqslant m$；其二是选择合适的散列函数。

这只适用于$|U|$较小，且关键字均事先已知的情况，此时经过精心设计散列函数 h 有可能完全避免冲突。

③ 冲突不可能完全避免

通常情况下，h 是一个压缩映像。虽然$|K|\leqslant m$，但$|U|>m$，故无论怎样设计 h，也不可能完全避免冲突。因此，只能在设计 h 时尽可能使冲突最少。同时还需要确定解决冲突的方法，使发生冲突的同义词能够存储到表中。

④ 影响冲突的因素

冲突的频繁程度除了与 h 相关外，还与表的填满程度相关。

设 m 和 n 分别表示表长和表中填入的结点数，则将 $\alpha=n/m$ 定义为散列表的装填因子(Load Factor)。α 越大，表越满，冲突的机会也越大。通常取 $\alpha\leqslant 1$。

2. 散列函数的构造

(1) 基本思想

散列函数的基本思想是：在记录的存储位置和关键字之间确定一种对应关系 f，使每一个关键字和结构中的一个存储位置相对应，由此只要知道关键字 K，就可以知道关键字的记录所存放的位置 $f(K)$，从而取得记录。

使用散列表，首先必须确定一个好的 HASH 函数，使关键字映像地址集合中任意地址的概率相同，即所得地址区间在整个地址区间中是随机的，称散列函数是均匀的。但是散列函数无论怎样，处理大量的数据时，仍然避免不了地址冲突(碰撞)的情况，所以建立比较好的 HASH 函数尤为重要。

建立 HASH 的方法有多种，但要注意的是 HASH 函数只对数值型的查找码转换，若为非数值型，需要用某种方法将它转换为数值型，再转换为地址。

① 除余法

取关键字被某一个不大于散列表表长 m 的质数 p 除后所得的余数为散列地址。

$$H(key)= key \bmod p \ (p \leqslant m)$$

其中 p 一般选择不大于 20 的质数。质数除余法是常用的构造 HASH 函数的方法之一。

② 平方取中法

计算 key 的平方，再取结果的中间部分值作为 HASH 地址。

③ 相乘取整法

该方法包括两个步骤：首先用关键字 key 乘以某个常数 $A(0<A<1)$，并抽取 key·A 的小数部分，然后用 m 乘以该小数后取整。

④ 随机数法

$$h(k)=\text{random}(key)$$

选取一个随机函数，取关键字的随机函数的值为散列地址。

(2) 碰撞(冲突)的处理法

通常来说，处理碰撞的方法有 3 种：开放地址法、拉链法和差值法。

开放地址法是将所有结点均存储在散列表 $T[0\cdots m-1]$中；拉链法是将互为同义词的结点链成一个单链表，只将链表的头指针存放在散列表 $T[0\cdots m-1]$数组中。

① 开放地址法

当冲突发生时，使用某种探查(探测)技术在散列表中形成一个探查(测)序列，沿此序列逐个单元地查找，直到找到给定的关键字或者碰到一个开放的地址(即该地址单元为空)为止。

$$Hi=(H(key)+di) \bmod m \qquad i=1，2，\cdots，k(k \leqslant m-1)$$

其中：$H(key)$为 HASH 函数，m 为散列表长；di 为增量序列，可有下列三种取法。

- di=1，2，3，…，$m-1$，称为线性探测再散列

基本思想：将散列表 $T[0\cdots m-1]$看成一个循环向量，若初始探查的地址为 d(即 $h(key)=d$)，则最长的探查序列为：

d，$d+1$，$d+2$，…，$m-1$，0，1，…，$d-1$

即依次从 $T[d]$开始探查，直到探查到 $T[d-1]$为止。

探查过程终止于 3 种情况：若当前探查的单元为空，表示查找失败，若是插入则将 *key* 写入其中；若当前探查的单元中含有 *key*，则查找成功，但对于插入意味失败；若探查到 $T[d-1]$时仍未发现空单元，也未找到 *key*，则是查找插入失败(因此时表满)。

线性探查法的探查序列表达式为：

$$hi=(h(key)+i)\ \%\ m \qquad 0\leqslant i\leqslant m-1$$

此时 $di=i$。

- $di=12，-12，22，-22，\cdots，\pm k^2(k\leqslant m/2)$，称为二次探测再散列

该方法的探查序列是：

$$hi=(h(key)+i^2)\ \%m \qquad 0\leqslant i\leqslant m-1$$

此时 $di=i^2$。

该方法的缺点是不易探查到整个散列空间。

- 双重散列法

该方法是开放地址法中的最好方法之一，它的探查序列为：

$$hi=(h(key)+i * hl(key))\ \%m \qquad 0\leqslant i\leqslant m-1$$

此时 $di=i * hl(key)$。

探查序列为：$d=h(key)$，$(d+hl(key))\%m$，$(d+2hl(key))\%m$ 等。该方法使用了 $h(key)$和 $hl(key)$ 两个散列函数，所以称为双重散列法。

定义 $hl(key)$的方法较多，但是无论采用什么方法定义，都必须使 $hl(key)$的值和 m 互素，才能使发生冲突的同义词地址均匀地分布在整个表中，否则可能造成同义词地址的循环计算。

若 m 为素数，则 $hl(key)=key\%(m-2)+1$。

② 拉链法

拉链法解决冲突的作法是，将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为 m 个链表，所有链表头存放在数组 $T[0\cdots m-1]$中，凡是散列地址为 i 的结点，均插入到以 $T[i]$为头指针的单链表中。T 中各分量的初值均应为空。

用拉链法处理碰撞时，散列表的每一个结点增加一个链接结点字段，用来连接同义词字表。

与开放地址法相比，拉链法有如下几个优点：

- 拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短。
- 由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况。
- 开放地址法为减少冲突要求装填因子 α 较小，故当结点规模较大时会浪费很多空间，而拉链法中可取 $\alpha \geq 1$，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间。
- 在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。而对开放地址法构造的散列表，删除结点不能简单地将被删结点的空间置为空，否则将截断在它之后填入散列表的同义词结点的查找路径，这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作时，只能在被删结点上做删除标记，而不能真正删除结点。

拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放地址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放地址法中的冲突，从而提高平均查找速度。

③ 差值法

差值法所采用的就是当散列函数产生的数据地址已有数据存在时，发生散列碰撞。处理的原则以现在的数据地址加上一个固定的差值，当数据地址超出数组大小时，让数据地址采用循环的方式处理，即新数据地址对数组大小取余数。

3. 散列表的算法

散列表上的运算有查找、插入和删除。其中主要是查找，这是因为散列表主要用于快速查找，且插入和删除均要用到查找操作。下面就是运用余数法作为散列函数及差值法解决散列碰撞问题的查找算法。

```
//=====================Program Discription=====================================
//程序名称：hsearch.java
//程序目的：运用余数法作为散列函数及差值法解决散列碰撞问题。
//==============================================================================
import java.util.*;
import java.io.*;
public class hsearch
{       public static int Max=6;                              //数据最大个数
 public static int HashMax=5;                                 //散列表最大个数
 public static int[] HashTab=new int[HashMax];                //散列表数组
      public static int[] Data=
              { 12,160,219,522,9997};                         //数据数组
```

```
    public static int Counter=1;                    //计数器
public static void main(String args[])
    {   int KeyValue;                               //欲查找值
    int Index;                                      //输入数组下标
    int i;                                          //循环变量
    Index=0;                                        //初始数组下标
      System.out.print("Input Data:");               //输出输入数据
    for (i=0;i<Max ;i++ )
          System.out.print("["+Data[i]+"]");
    System.out.println("");
    for (i=0;i<HashMax ;i++ )                       //散列表初始化
        HashTab[i]=0;
    while (Index<Max)                               //建立散列表
    {   if (CreateHash(Data[Index]))
          System.out.println("Hash Success!!");     //建立成功
        else
              System.out.println("Hash Fulled!!"); //建立失败
          Index++;
    }
      for (i=0;i<HashMax ;i++ )                     //输出散列表数据
        System.out.print("["+HashTab[i]+"]");
      System.out.println("");
        ConsoleReader console=new ConsoleReader(System.in);
while (true)      //利用散列表查找数据
    {system.out.print("Please enter your key value(0 for Exit):");
InputStreamReader is=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(is);
    StringTokenizer st;
      try{  String myline=br.readLine();
            st=new StringTokenizer(myline);
            KeyValue=Integer.parseInt(st.nextToken());//取得输入值
          }
      catch(IOException ioe)
          { System.out.print("IO error:"+ioe);
            }
      if (KeyValue==0)                  //若输入 0 则结束程序
            break;
      if (HashSearch(KeyValue))       //输出查找次数
            System.out.println("Search Time="+ Counter);
      else        //输出没有找到数据
            System.out.println("No Found!!");
}}
```

```
//-----------------------------------------------------
//散列函数中的余数法
//-----------------------------------------------------
    public static int HashMod(int KeyValue)
 {          return KeyValue % HashMax;      //返回的键值除以散列表大小取余数
 }
//-----------------------------------------------------
//差值法
//-----------------------------------------------------
    public static int CollisionOffset(int Address)
 {    int Offset=3;      //设差值为 3
      return (Address+Offset) % HashMax;     //返回旧地址加差值除以散列表大小取余数
 }
//-----------------------------------------------------
//建立散列表
//-----------------------------------------------------
    public static boolean CreateHash(int KeyValue)
 {
      int HashTime;       //散列次数
      int CollisionTime; //碰撞次数
          int Address;     //数据地址
int i;                    //循环变量
HashTime=0;       //初始散列次数
   CollisionTime=0; //初始碰撞次数
Address=HashMod(KeyValue);       //调用散列函数
while (HashTime<=HashMax)        //散列次数超过散列表容量
{                   if (HashTab[Address]==0)
      { HashTab[Address]=KeyValue;                    //可存储数据
           System.out.print("Key:"+KeyValue);     //输出位置
         System.out.println("=>Address"+Address);
           for (i=0;i<HashMax ;i++ )                  //输出散列表内容
         { System.out.print("["+HashTab[i]+"]");
         }
        System.out.println("");
         return true;
      }
      else                                 //不可存储数据
      {    CollisionTime++;        //累计碰撞次数
           System.out.print("Collision"+CollisionTime);
           System.out.println("=>Address"+Address);
           Address=CollisionOffset(Address);      //调用差值法
      }
      HashTime++;                                    //累计散列次数
}
```

```
        return false;
    }
//-----------------------------------------------------
//散列查找法
//-----------------------------------------------------
    public static boolean HashSearch(int KeyValue)
    {   int Address;                                    //数据地址
        Counter=0;
        Address=HashMod(KeyValue);                      //调用散列函数
        while (Counter<HashMax)
        {   Counter++;
            if (HashTab[Address]==KeyValue)             //找到数据
                return true;
            else                                        //未找到数据
                Address=CollisionOffset(Address);       //调用差值法
        }
        return false;
}}
```

4. 性能分析

由于插入和删除的时间均取决于查找，下面只分析查找操作的时间性能。

虽然散列表在关键字和存储位置之间建立了对应关系，理想情况是无须关键字的比较就可找到待查关键字。但是由于冲突的存在，散列表的查找过程仍是一个和关键字比较的过程，不过散列表的平均查找长度比顺序查找、二分查找等完全依赖于关键字比较的查找要小得多。

一般情况下，处理冲突(碰撞)方法相同的散列表，其平均查找长度依赖于散列表的装填因子。

散列表的装填因子的定义为：

$$\alpha = \text{表中填入的记录数/散列表的长度}$$

α 标志散列表的装满程度。直观地讲，其值越小，发生冲突的可能性就越小；否则，发生冲突的可能性就越大。

(1) 查找成功的 ASL

散列表上的查找优于顺序查找和二分查找。

线性探查法查找成功的平均查找长度为：

$$ASL \approx [1+1/(1-\alpha)]/2$$

拉链法查找成功的平均查找长度为：

$$ASL \approx 1+\alpha/2$$

(2) 查找不成功的 ASL

对于不成功的查找，顺序查找和二分查找所需进行的关键字比较次数仅取决于表长，而散列查找所需进行的关键字比较次数和待查结点有关。因此，在等概率情况下，也可将散列表在查找不成功时的平均查找长度定义为查找不成功时对关键字需要执行的平均比较次数。

需注意的是：

① 由同一个散列函数、不同的解决冲突方法构造的散列表，其平均查找长度是不相同的。

② 散列表的平均查找长度不是结点个数 n 的函数，而是装填因子 α 的函数。因此在设计散列表时可选择 α 以控制散列表的平均查找长度。

③ α 的取值。α 越小，产生冲突的机会就小，但 α 过小，空间的浪费就过多。只要 α 选择合适，散列表上的平均查找长度就是一个常数，即散列表上查找的平均时间为 $O(1)$。

④ 散列法与其他查找方法的区别。除散列法外，其他查找方法的共同特征为：均是建立在比较关键字的基础上。其中顺序查找是对无序集合的查找，每次关键字的比较结果为“=”或“!=”两种可能，其平均时间为 $O(n)$；其余的查找均是对有序集合的查找，每次关键字的比较有“=”、“<”和“>”3 种可能，且每次比较后均能缩小下次的查找范围，故查找速度更快，其平均时间为 $O(\lg n)$。而散列法是根据关键字直接求出地址的查找方法，其查找的期望时间为 $O(1)$。

思考和练习

(1) 有一组数据，内容如下：

3,15,36,57,66,88,99,100,123,134,256

试用顺序查找法查找 88 和 300，写出运作过程。

(2) 有一组数据，内容如下：

8,15,38,57,68,88,98,108,129,234,256

试用二分查找法查找 88 和 222，写出运作过程。

(3) 有一组数据，内容如下：

8,15,6,57,99,88,98,18,12,234,256

试建立其二叉排序数，并列表算出各数据的查找次数。

(4) 试运用平方取中法作为散列函数及拉链法来设计散列表。

(5) 实现一个类，其功能是统计一个文本文件中单词的频率，输出这些单词及其频率的列表。

第9章　动态存储管理

数据结构包括了存储结构和逻辑结构，只有在确定了存储结构的情况下，可以实现具体算法。但是在数据结构中只是借助高级语言加以描述，并没有涉及具体的存储分配。本章主要讨论存储空间的分配和管理方法。

本章的学习目标：

- 动态存储管理的分配方法；
- 存储空间的回收方法；
- 存储紧缩。

9.1　概　　述

在介绍的 3 种数据结构——线性结构、层次结构和网状结构中，使用高级语言描述了它们的内存映像但并没有涉及具体的存储分配。实际上结构中的每个数据元素都占有一定的内存位置，在程序的执行过程中，数据元素的存取是通过对应的存储单元来进行的。

在早期的计算机上，这个存储管理的工作是由程序员自己来完成的。在程序执行之前，首先需将用机器语言或汇编语言编写的程序输送到内存的某个固定区域上，并预先给变量和数据分配好对应的内存地址(绝对地址或相对地址)。有了高级语言之后，程序员不需要直接和内存地址打交道，程序中使用的存储单元都由逻辑变量(标识符)来表示，它们对应的内存地址都是由编译程序在编译或执行时进行分配。

另一方面，当计算机是被单个用户使用时，整个内存除操作系统占用一部分之　　外，都归这个用户的程序使用(如 PDP-11/01 的内存为 32KB，系统占用 4KB，用户程序可用 28KB)。但在多用户分时并发系统中，多个用户程序共享一个内存区域，此时每个用户程序使用的内存就由操作系统来进行分配了。并且，在总的内存不够使用时，还可采用自动覆盖技术。

对操作系统和编译程序来说，存储管理都是一个复杂而又重要的问题。不同语言的编译程序和不同的操作系统可以采用不同的存储管理方法。它们采用的具体做法，读者将在操作系统中学习。本课程仅就动态存储管理中涉及的一些基本技术进行讨论。

动态存储管理的基本问题是系统如何应用户提出的“请求”分配内存？又如何回收那些用户不再使用而“释放”的内存，以备新的“请求”产生时重新进行分配？提出请求的用户可能是进入系统的一个作业，也可能是程序执行过程中的一个动态变量。因此，在不

同的动态存储管理系统中，请求分配的内存量大小不同。通常在编译程序中是一个或几个字，而在系统中则是几千、几万，甚至是几十万字。然而，系统每次分配给用户(不论大小)的都是一个地址连续的内存区。为了叙述方便，在下面的讨论中，将统称已分配给用户使用的地址连续的内存区为“占用块”，称未曾分配的地址连续的内存区为“可利用空间块”或“空闲块”。

显然，不管什么样的动态存储管理系统，在刚开始时，整个内存区是一个“空闲块”。随着用户进入系统，并先后提出存储程序或数据的请求，系统则依次进行分配。因此，在系统运行的初期，整个内存区基本上分隔成两大部分：低地址区包含若干已经被程序或数据使用的占用块；高地址区(即分配后的剩余部分)是内存中没有被分配程序和数据的部分，是一个“空闲块”。

例如图 9-1 所示为依次给 8 个用户进行分配后的系统的内存状态。经过一段时间以后，有的用户运行结束，它所占用的内存区变成空闲块，这就使整个内存区呈现出占用块和空闲块交错的状态，如图 9-2 所示。

图 9-1　动态存储分配过程中的初始状态

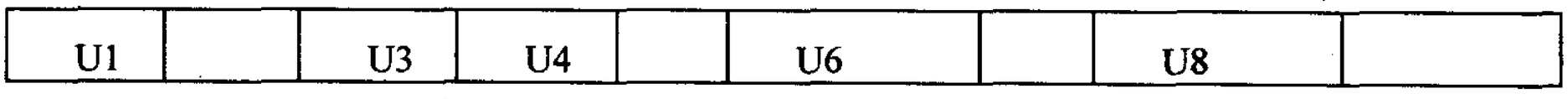


图 9-2　系统运行若干时间后的内存分配图

在计算机运行长时间后，可能出现一个新的用户，申请的空间大小超过 U1 和 U3 之间的空间大小，此时可能所有的小块的空间都不能满足新用户的要求，但是所有的小空间的总和却大于新用户所需要的空间，这就造成新用户不能得到内存分配的空间而无法运行，使新用户处于等待状态。解决方案相对容易，如将所有的数据移动，使所有的小空闲块连接成一个大空闲块，然后分配给新的用户，但是此时，可能需要大量的数据移动，这就占用了大量的 CPU 时间，造成系统的效率降低。

9.2　内存分配与回收策略

程序的执行离不开内存，因为 CPU 不能和外存进行数据交换，它只能和内存进行数据交换，所以内存的分配与回收是内存管理的主要功能之一。无论采用哪一种管理和控制方式，能否把外存中的数据和程序调入内存，取决于能否在内存中为它们安排合适的位置(内存空间够用)。因此，存储管理模块要为每一个同时执行的程序分配内存空间。另外，当程序执行结束之后，存储管理模块又要及时回收该程序所占用的内存资源，以便将空间分配给其他程序。

为了有效合理地利用内存，设计内存的分配和回收方法时，必须考虑和确定以下几种策略和数据结构。

(1) 分配结构：需要登记内存使用情况，供分配程序使用的表格与链表。例如内存空闲区表、空闲区队列等。分配结构直接影响到查找可用空闲块的时间，同时也影响到内存的回收效率。

(2) 放置策略：确定调入内存的程序和数据在内存中的位置。这是一种选择内存空闲区的策略。该策略的好与坏可以直接影响内存的使用效率和内存分配时查找可用空闲块的时间。

(3) 交换策略：在需要将某个程序段和数据调入内存时，如果内存中没有足够的空闲区，由交换策略来确定把内存中的哪些程序段和数据段调出内存，以便腾出足够的空间。该调度方法可以提高内存的使用效率。

(4) 调入策略：外存中的程序段和数据段什么时间按什么样的控制方式进入内存。调入策略与内外存数据流动控制方式有关。

(5) 回收策略：回收策略包括 2 点，一是回收的时机，二是对所回收的内存空闲区和已存在的内存空闲区的调整。

实际上，对内存的管理或对内存的分配策略通常有两种方法：

(1) 系统从高地址的空闲块中进行分配空间，直到分配无法进行时，系统才去回收所有用户不再使用的空闲块，重新组织内存，并将所有的空闲块连接起来使之形成一个大的空闲块，又可以继续分配空间。

(2) 用户一旦运行结束，立即将他所占用的内存释放，使之变成空闲块，当新的用户申请空间时，搜寻所有的空闲块，找到最合适的空闲块分配给他。此时需要建立一个空闲块的链表或目录表，也称为“可利用空闲表”，由该表统一管理所有的空闲空间。在目录表中，每个表目应该包含的信息有初始地址、空闲块大小和使用情况；在链表中，每个结点表示一个空闲块，系统每次分配或回收一块空间相当于在空闲链表中删除或插入一个空闲结点。

选用何种方法，一般由操作系统确定。在数据结构中，一般选择第 2 种分配方案讨论内存管理。

9.3　可利用空间的分配方法

目录表的情况将在操作系统课程中详细介绍，在此仅就链表的情况进行讨论。链表情况是指内存的空闲块，通过单向链表或双向链表连接起来。

如上所述，可利用空间表中包含所有可分配的空闲块，每一块是链表中的一个结点。当用户请求分配时，系统从可利用空闲表中删除一个结点分配之；当用户释放其所占内存时，系统即回收并将它插入到可利用空闲表中。因此，可利用空闲表亦称作“存储池”。根据系统运行的不同情况，可利用空闲表可以有下列 3 种不同的结构形式。

第 1 种情况，系统运行期间所有用户请求分配的存储量大小相同。对此类系统，通常

的做法是，在系统开始运行时将归它使用的内存区按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。由于表中结点大小相同，则分配时无需查找，只要将第一个结点分配给用户即可；同样，当用户释放内存时，系统只要将用户释放的空闲块插入在表头即可。可见，这种情况下的可利用空闲表实质上是一个链栈。这是一种最简单的动态存储管理的方式。但是此种情况在分配空间时，有可能浪费比较多的存储空间，使空间的利用率大大降低。例如，一段程序可能需要的空间非常小，它也需要从链表中删除一个结点，此时结点空间浪费比较大。同时此种分配方案对于多道程序的执行不能够根据需要分配，而只能无论大小程序或数据都分配相同大小的空间，对于较大的程序可能会造成无法分配空间，导致程序无法运行。

第 2 种情况，系统运行期间用户请求分配的存储量有若干种大小的规格。对此类系统，一般情况下是建立若干个可利用空间表，同一链表中的结点大小相同。例如，某动态存储管理系统中的用户将请求分配 2 个字、4 个字或 8 个字的内存块，则系统建立三个结点大小分别为 3 个字、5 个字和 9 个字的链表，它们的表头指针分别为 av2、av4 和 av8。每个结点中的第一个字设有链域(link)、标志域(tag)和结点类型域(type)。结点结构如图 9-3 所示，其中：类型域为区别 3 种大小不同的结点而设，type 的值为 0、1 或 2，分别表示结点大小为 2 个字、4 个字或 8 个字；标志域 tag 为 0 或 1 分别表示结点为空闲块或占用块；链域中存储指向同一链表中下一结点的指针，而结点中的值域是其大小分别为 2、4 和 8 个字的连续空间。此时的分配和回收的方法在很大程度上和第 1 种情况类似，只是当结点大小和请求分配的量相同的链表为空时，需查询结点较大的链表，并从中取出一个结点，将其中一部分内存分配给用户，而将剩余部分插入到相应大小的链表中。回收时，也只要将释放的空闲块插入到相应大小的链表的表头中去即可。然而，这种情况的系统还有一个特殊的问题要处理：即当结点与请求相符的链表和结点更大的链表均为空时，分配不能进行，而实际上内存空间并不一定不存在所需大小的连续空间，只是由于在系统运行过程中，频繁出现小块的分配和回收，使得大结点链表中的空闲块被分隔成小块后插入在小结点的链表中，此时若要使系统能继续运行，就必须重新组织内存，即执行“存储紧缩”的操作。

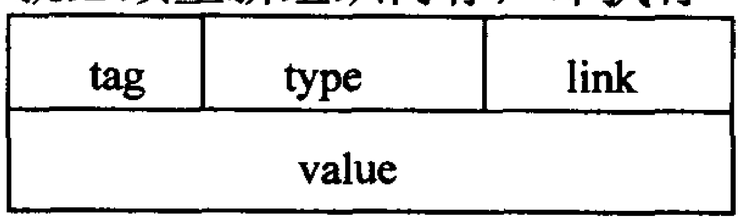


图 9-3　空闲块结构

$$tag = \begin{cases} 0 & \text{空闲块} \\ 1 & \text{占用块} \end{cases}$$

$$type = \begin{cases} 0 & \text{结点大小为 2 个字} \\ 1 & \text{结点大小为 4 个字} \\ 2 & \text{结点大小为 8 个字} \end{cases}$$

第 3 种情况，系统在运行期间分配给用户的内存块的大小不固定，可以随请求而变。因此，可利用空间表中的结点即空闲块的大小也是随意的。通常，操作系统中的可利用空间表属于这种类型。

系统刚开始工作时，整个内存空间是一个空闲块，即可利用空间表中只有一个大小为整个内存区的结点，可利用空间表的大小和个数随着分配和回收的不停进行而发生变化。

实际上，在内存分配过程中，链表中的结点大小不同，所以在结点结构的描述上还需要增加存储空间的大小域 size，用 size 描述空闲块的存储量。当然要求每个结点描述的空间是地址连续的存储空间(如图 9-4 所示)，但是当需要将空闲块合并时，特别是前面有相邻的空闲块，需要能够查询当前块是否是空闲，在空闲块的末尾增加一个 tag 位，还需要能够从末尾位置直接找到当前块的首地址，因此在末尾还需要增加一个指针，指向该空闲块的首地址(uplink)。

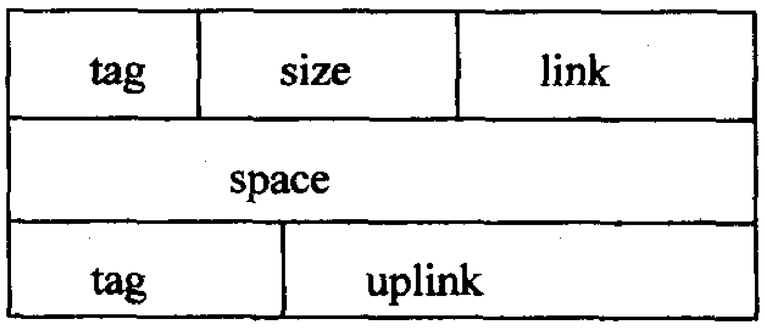


图 9-4　空闲表中大小不定的结点结构

$$Tag=\begin{cases}0 & \text{空闲块}\\ 1 & \text{占用块}\end{cases}$$

size 表示空闲块的大小，link 指针指向下一个空闲块，uplink 指针指向该空闲块的首地址。

此时分配空间时，如果空闲块的大小是 n，程序需要分配 m 个存储空间，此时分配一个结点时还剩下 $n-m$(假设 $n>m$)个空间，而 $n-m$ 个空间仍然存在空闲表中，对应的 m 个空间的标志位为 1，表示已经被占用；而 $n-m$ 个空闲空间的标志位为 0，表示当前块仍然是空闲块。值得注意的是，标志位有两个，首标志位和末尾标志位。因为查找空间时，总是需要的空间要小于将要分配的空间，可能会出现分配完空间后，剩余的空间不能够被其他的程序或数据使用，这些小的空闲块称为“碎片”。如果剩余若干个比较小的空闲块时，可能造成空间不能用的情况。通常采用 3 种不同的分配策略。

(1) 首次拟合法

首次拟合法的基本思想是：从表头指针开始查找可利用空间表，将找到的第一个大小不小于 n 的空闲块的一部分分配给用户。可利用空间表本身既不按结点的初始地址有序，也不按结点的大小有序。则在回收时，只要将释放的空闲块插入在链表的表头即可。

首次拟合法明显有以下两个缺点：

- 系统工作一段时间后，空闲表中剩下的空闲块可能都是一些零碎的小空闲块，即产生“碎片”，而这些“碎片”又不足以为其他的任何程序或数据所利用，造成空间的大量浪费。

- 为了查找一个适合用户的空闲块，每次需要从链表的表头开始进行搜索，而那些比较小的空闲块往往集中在链表的前部，查找的效率非常低。

鉴于首次拟合法的不足，可将算法做如下一些改进：

- 将可用链表改成双向循环链表。这样，删除某链结点时很容易找到其直接前趋结点。
- 将固定的头指针改为活动指针。每次分配以后那些零碎自由块都集中在表的前部，只要将刚刚分配的结点作为下次搜索的起始位置，这样可以向后搜索，从而提高查找效率。
- 确定一个 ε，当某个自由块分配后剩下的内存不足 ε 时，便将该自由块全部分配给某段程序，并将其从空闲链表中删除。
- 当分配块返回可用表时，先检查其物理上邻接的存储块是否在可用空闲表中。若在表中，则把释放的分配块合并于该物理邻接的空闲块；否则，释放的分配块作为一个新的空闲块插入到可用空闲表中。
- 为了便于判断物理上邻接的存储块是否是空闲块，在分配块与空闲块的头、尾各设置一个标志 TAG，用来表示该块是否为空闲块。

(2) 最佳拟合法

最佳拟合法的基本思想是：将空闲区的空闲块按从小到大的顺序组成自由链，当用户申请一个空闲区大小为 n 时，将在可利用空间表中找到一个大于 n 的最小结点分配给用户，(最适合用户需要的空间的空闲块)使空间的浪费减少到最少。如果 n 的值刚刚大于需要的空间，剩余的空间又不能被其他的程序或数据使用，容易产生“碎片”。

如果空闲区的大小大于请求长度，则减去请求长度 n 后，剩余的空闲区部分仍然按照剩余的大小插入在空闲表中(需要在删除结点后，查找插入的位置，然后将结点插入到链表合适的位置中)。这样最佳拟合法总是保证了空闲区中的空闲块按照从小到大排列。但是浪费了大量的时间。

最佳拟合法的缺点是：总是可能产生碎片，造成大量的碎片后，可能形成某个申请空间分配不到，但是所有的碎片总和能够满足该申请；查找时间延长，因为自由链中的空闲块按照从小到大排列，每次分配总是从头到尾查找，造成查找时间延长。

解决方案：一旦剩余的空间小于某个特定的值，此时将它直接分配给程序或数据即可，不会产生“碎片”；解决时间问题，可以采用双向链表，既可以向后查找，又可以向前查找，减少从头扫描时间；或者采用活动的头指针。

(3) 最差拟合法

与最佳拟合法相反，最差拟合法是将所有的空闲块按照从大到小的顺序存储，尽量避免产生碎片。最差拟合法的基本思想是：将在可利用的空间表中找到一个最大的空闲块分配给用户，使得分配空间尽可能地满足用户的需要。

当用户申请空间时，总是从当前最大的空闲块中划分空间给用户，空间大小减去请求长度 n 后，剩余的空闲区部分仍然按照剩余的大小插入在空闲表中(需要在删除结点后，查找插入的位置，然后将结点插入到链表合适的位置中)。这样最差拟合法总是保证了空闲区

中的空闲块按照从大到小排列。

最差拟合法的缺点是：总是分配最大的空间，使最大空间逐渐变小，可能导致某个用户申请的空间不能够得到满足。

解决方案是：使用最佳拟合法或者释放空间时，尽量将相邻空间合并为更大的空间，保证大程序的运行，或者采用活动的头指针以减少查找时间。

(4) 几种分配算法的比较

上面讨论了 3 种常用的内存分配算法及回收算法。由于回收后的空闲区要插入可用表或自由链中，而且可用表或自由链是按照一定顺序排列的。所以，除了搜索查找速度与所找到的空闲区是否最佳，释放空闲区的速度也对系统开销产生影响。下面从查找速度、释放速度及空闲区的利用等 3 个方面对上述 3 种算法进行比较。

首先，从搜索速度上看，首次拟合法具有最佳性能。尽管最佳拟合法或最差拟合法看上去能很快地找到一个最适合的或最大的空闲区，但后两种算法都要求首先把不同大小的空闲区按其大小进行排队，这实际上是对所有空闲区进行一次搜索。再者，从回收过程来看，首次拟合法也是最佳的。因为使用首次拟合法回收某一空闲区时，无论被释放区是否与空闲区相邻，都不用改变该区在可用表或自由链中的位置，只需修改其大小或起始地址。而最佳拟合法和最差拟合法都必须重新调整该区的位置。

首次拟合法的另一个优点就是尽可能地利用了低地址空间，从而保证高地址有较大的空闲区来放置要求内存较多的进程或作业。

反过来，最佳拟合法找到的空闲区是最佳的，也就是说，用最佳拟合法找到的空闲区或者是正好等于用户请求的大小或者是能满足用户要求的最小空闲区。不过，尽管最佳拟合法能选出最适合用户要求的可用空闲区，但这样做在某些情况下并不一定提高内存的利用率。例如，当用户请求小于最小空闲区不太多时，分配程序会将其分配后的剩余部分作为一个新的小空闲区留在可用表或自由链中。这种小空闲区(碎片)有可能永远得不到再利用(除非与别的空闲区合并)，而且也会增加内存分配和回收时的查找负担。

最差拟合法正是基于不留下碎片空闲区这一出发点的(为解决最佳拟合法的缺点)。它选择最大的空闲区来满足用户要求，以期分配后的剩余部分仍能进行再分配，但是对于大的空闲块的不断划分，容易引起要求大容量内存的进程或作业不能满足的情况。

总之，上述 3 种算法各有特长，针对不同的请求队列，效率和功能是不一样的。

因此不同的情景需要采用不同的方法，通常在选择时需要考虑以下因素：用户的逻辑要求；请求分配量的大小分布；分配和释放的频率以及效率对系统的重要性等。

在实际使用的系统中回收空闲块时，还需要考虑一个结点的合并问题。因为在系统不断地进行分配和回收的过程中，大的空闲块逐渐被划分为小的占用块，在它们重新成为空闲块回收后，即使地址相邻的两个空闲块也要作为两个结点插入到可利用空间表中，导致后来出现的大容量的请求分配无法进行。为了更加有效地利用内存，就要求系统在回收时应考虑将地址相邻的空闲块合并成尽可能大的结点。也就是说，在回收空闲块时，应首先检查地址与它相邻的内存是否是空闲块，如果是就将它们合并后，作为一个空闲块存在自

由链中，否则只要将空闲块直接插入到自由链中即可。

9.4 存 储 紧 缩

动态存储管理内存，可以采用另外一种动态存储管理——存储紧缩。具体方法是：在整个动态存储管理过程中，不管哪个时刻，可利用空间都是一个连续的存储区，在编译过程中称为“堆”，每次分配都是从这个可利用空间中划出一块，释放(回收)时，必须将回收的空闲块合并到整个堆中，才能够重新使用，也就是存储紧缩。

堆的实现方法：设立一个指针，称为堆指针，始终指向堆的最低(或最高)地址。当用户申请 N 个存储块时，堆指针向高地址(或低地址)移动 N 个存储单位，移动之前的指针就是分配给用户的占用块的初始地址。

因为可利用空间是连续的，所以利用堆指针实现地址分配非常方便，但是回收用户释放的空间就比较麻烦。一般有两种方法：

- 一旦有用户释放存储块即进行回收紧缩；
- 在程序执行过程中不回收用户随时释放的存储块，直到可利用空间不够分配或堆指针指向最高地址时才进行存储紧缩。

存储紧缩的目的是将堆中所有的空闲块连成一片。

为实现存储紧缩，首先要对占用块进行“标志”，标志方法和 9.3 节介绍的类同(存储块的结构可能不同)；其次需进行下列 4 步操作：

- 计算占用块的新地址。从最低地址开始巡查整个存储空间，对每一个占用块找到它在紧缩后的新地址。为此，需设立两个指针随巡查向前移动，这两个指针分别指示占用块在紧缩之前和之后的原地址和新地址。因此，在每个占用块的第一个存储单位中，除了设立长度域(存储该占用块的大小)和标志域(存储区别该存储块是占用块或空闲块的标志)之外，还需设立一个新地址域，以存储占用块在紧缩后应有的新地址，即建立一张新、旧地址的对照表。
- 修改用户的初始变量表，以便在存储紧缩后用户程序能继续正常运行。
- 检查每个占用块中存储的数据。若有指向其他存储块的指针，则需作相应地修改。
- 将所有占用块迁移到新地址去。

至此，完成了存储紧缩的操作。最后，将堆指针赋以新值(即紧缩后的空闲存储区的最低地址)。可见，存储紧缩法不仅要传送数据(进行占用块迁移)，而且要修改所有占用块中的指针值。因此，存储紧缩是一个系统操作，除非迫不得已，否则就不用。

存储紧缩时，一般有 4 种情况可以考虑：

- 释放的空间前后都没有相邻的空闲块；
- 释放的空间前有相邻的空闲块；
- 释放的空间后有相邻的空闲块；

● 释放的空间前后都有相邻的空闲块。

以上 4 种情况能够合并空间的尽量合并空间，不能够合并的则直接插入到空闲表中即可。

对于第 1 种情况，直接将结点的标志位 tag 置为 0(注意是两个标志位)，然后插入到空闲表中即可。

对于第 2 种情况，假设将要释放的结点是 p，只要检查当前结点 $p-1$ 位置上的标志位是否等于 0 即可，如果等于 0，说明该结点前面的结点是空闲块，可以将结点与其前面的结点合并。合并方法是：找到 $p-1$ 位置上的 uplink，可以找到前面结点的首地址，假设是 q，将 q 的 size 改为两个结点的 size 之和。通过将 p 结点的末尾标记位($p+size-1$ 的 tag)置为 0，同时将 $p+size-1$ 地址上的 uplink 改为 q(合并新结点的首地址)，完成了两个结点的合并。

对于第 3 种情况，假设将要释放的结点是 p，只要检查当前结点 $p+size$ 位置上的标志位是否等于 0 即可，如果等于 0，说明该结点后面的结点是空闲块，可以将结点与其后面的结点合并。合并方法是：找到 $p+size$ 的地址，假设为 q，通过 q 可以找到 $q+size-1$ 对应结点的 uplink，将此 uplink 的值改为 p，同时将 p 的 size 改为两个结点的 size 之和，将 p 的标志位 tag 改为 0 即可。

对于第 4 种情况，既要检查 p－1 的标志位是否为 0，又要检查 p+size 的 tag 位是否为 0。如果都为 0，说明该结点前、后面的结点是空闲块，可以将结点与其前、后面的结点合并。合并方法同第 2 种情况，只是在修改 uplink 时，应该是后面结点的 uplink。同时 size 应该是 3 个结点的 size 之和。

以上 4 种情况，只要使用 switch 语句，分开处理即可。

思考和练习

(1) 动态存储管理中内存分配方法通常有几种？比较它们的优缺点。

(2) 写出首次拟合法流程框图。

(3) 针对不同的内存分配方法的缺点，设计相应的存储结构解决它。

(4) 在不同的分配方法中，说明回收空闲块的方法。

(5) 说明紧缩存储与常用分配方法的不同。

第10章　文 件 管 理

文件是大量记录的集合，一般把主存储器(内存储器)中的记录集合称为表，把存储在外存储器中的记录集合称为文件。本章讨论在外存储器中的文件的表示方法及其运算的实现方法。

本章的学习目标：

- 文件存储器；
- 文件的逻辑结构；
- 索引文件；
- 散列文件。

10.1　文件的基本概念

10.1.1　文件定义

算法和数据结构的实现既可以基于主存储器，也可以基于辅助存储器，在前面的章节中通常考虑使用的是主存储器，但使用不同的存储器(主、辅助存储器)会直接影响算法和数据结构的设计。文件是性质相同的记录的集合，它通常存储在外存(辅助存储器)上，本章主要考虑外存储器。

文件通常存储在外存(辅助存储器)上，对文件的操作就要考虑文件的存储介质问题，因为存储介质的不同，数据的访问速度、数据的存储量以及数据的存取方法等可能就不同，而以上的因素就可能决定了算法的实现。

操作系统的文件是一维的、连续的字符序列，无结构、无解释；它也是记录的集合，用户为了存取、加工方便，把文件中的信息划分成若干组，每一组信息称为逻辑记录，并且可按顺序编号。

数据库中的文件是带有结构的记录的集合。记录是由一个或多个数据项组成的集合，它也是文件存取的数据的基本单位。

在文件中常见的术语有：

- 记录　是文件中存取的基本单位，数据项是文件可使用的最小单位。
- 数据项　有时也称为字段，或者称为属性。

- 主关键字项　其值能惟一标识一个记录的数据项或数据项的组合。
- 次关键字项　其值不能惟一标识一个记录的数据项。
- 主关键字(或次关键字)　主关键字项(或次关键字项)的值称为主关键字(或次关键字)。

有时为描述方便，将主(或次)关键字项简称为主(或次)关键字，并且假定主关键字项只含一个数据项。

- 单关键字文件　文件中的记录只有一个惟一标志记录的主关键字。
- 多关键字文件　文件中的记录除有一个惟一标志记录的主关键字外，还含有若干个次关键字。

按照构成文件的记录结构的长度分为定长记录文件和不定长记录文件。

文件中记录含有的信息长度相同，称为定长记录，由定长记录组成的文件称为定长文件。若文件中记录含有的信息长度不等，则称为不定长文件。

10.1.2　文件逻辑结构及操作

记录的逻辑结构是指记录在用户或应用程序员面前呈现的方式，是用户对数据的表示和存取方式。

文件是性质相同的记录的集合，简单地说，文件可以被认为是每条记录为一行的二维表格，如果把记录简略成结点(此时可以以关键字作为记录的惟一标识)，那么文件中每个记录最多只有一个直接后继记录和一个直接前趋记录，而文件的第一个记录只有后继没有前趋，文件的最后一个记录只有前趋而没有后继。因此，文件可看成是一种线性结构。

记录的物理结构是数据在物理存储器上的存储方式，是数据的物理表示和组织。

文件的物理结构是指文件在存储介质上的组织方式。

记录的逻辑结构和物理结构的着眼点不同，逻辑结构主要是让用户使用方便，而物理结构则主要是提高存储空间的利用率和减少存取时间，它根据需要及设备本身的特性有很多种方式。

对应不同结构的记录也分别称为物理记录和逻辑记录，两种不同的记录是两个不同的概念。物理记录是指计算机使用一条 I/O 指令进行读/写的基本数据单位，对固定设备而言，它的大小基本上固定不变，而逻辑记录的大小则是由使用要求决定的，它们之间有以下关系：

- 一条物理记录存放一条逻辑记录。
- 一条物理记录存放多条逻辑记录。
- 多条物理记录存储一条逻辑记录。

用户在使用逻辑记录时，可以不必关心物理记录的存储位置，它们之间的地址转换以及存取由操作系统实现。

但用户在使用文件时必须对其进行必要的操作。对文件的操作有很多种，文件的操作

主要有检索和维护。检索是在文件中查找满足给定条件的记录。维护主要是对文件进行记录的插入、删除及修改等更新操作。

其中文件的检索有 3 种方式：

- 顺序存取　按记录号依次存取逻辑记录。
- 直接存取　按照记录号或记录的相对位置直接取得需要的记录。
- 按关键字存取　给定一个关键字的值，查询一个或多个关键字与给定值相关的记录，一般有 4 种查询方法。
  - ◊ 简单查询：查询关键字等于给定值的记录。例如查询学号是 00001 的记录。
  - ◊ 区域查询：查询关键字属于某个范围的记录。例如查询成绩在 80 分以上的所有记录。
  - ◊ 函数查询：给定关键字的值使函数成立的记录。例如查询所有男生的记录。
  - ◊ 布尔查询：通过布尔运算组合起来的查询。例如查询男生中成绩在 90 分以上的 2003 届的所有记录。

10.2　文件的分类

文件按不同的组织方式可以分为若干种，其基本组织形式分为顺序组织、索引组织、散列组织和链组织。对应文件的分类一般有顺序文件、索引文件、直接存取文件(散列文件)和多关键字文件等。按记录的特性来分类，可以将文件分为定长记录文件(文件中每条记录含有的信息长度相等)和不定长记录文件(文件中含有的信息长度不相等)。

10.2.1　顺序文件

记录按其在文件中的逻辑顺序依次存入存储介质所建立的文件称为顺序文件。顺序文件是根据记录的序号或记录的相对位置来进行存取的文件组织方式。

顺序文件中如果次序相连的两条记录在存储介质上的存储位置是相邻的，称为连续文件，反之称为串联文件。

其特点是：

- 存取第 I 个记录，必须先搜索第 $I-1$ 个记录。
- 插入新的记录时只能加在文件的末尾。
- 若要更新文件中的某个记录，则必须将整个文件进行复制。

顺序文件的优点是连续存取的速度快，因此主要用于只进行顺序存取、批量修改的情况。顺序文件的存储介质比较典型的是磁带。

10.2.2　索引文件

1. 索引文件概述

索引文件由索引区和文件数据区两部分组成，其中文件数据区按关键字有序的称为索引顺序文件；文件数据区中记录不按关键字顺序排列称为索引非顺序文件。索引非顺序文件通常是指索引文件。

索引区指明逻辑记录和物理记录之间一一对应关系，称为索引表。一般索引表由索引项组成，索引项一般有两部分：关键字和关键字对应的记录地址。

数据区和索引表构成索引文件(如图 10-1 所示)。

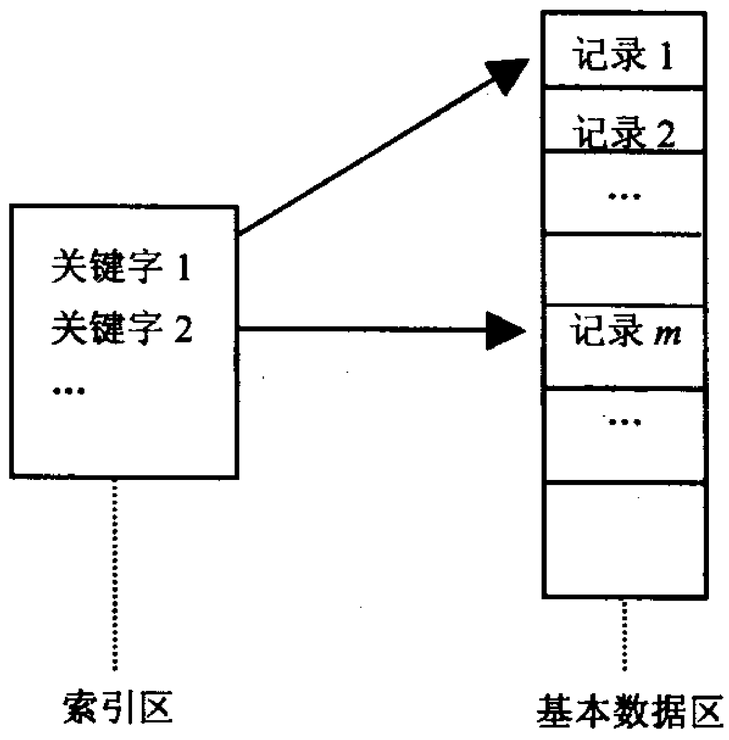


图 10-1　索引文件结构

根据关键字和记录地址是否一一对应可以将索引文件分为稠密索引和稀疏索引两类。

- 稠密索引：对于索引非顺序文件，由于主文件中记录是无序的，则必须为每个记录建立一个索引项，这样建立的索引表称为稠密索引。
- 稀疏索引：对于索引顺序文件，由于主文件中记录按关键字有序，则可对一组记录建立一个索引项，称为稀疏索引。

索引文件在存储器上分为两个区：索引区和数据区。前者存放索引表，后者存放主文件(数据)。

建立索引文件的主要目的是提高查询速度，对索引文件而言其检索步骤为：首先将外存上含有索引区的页块送入内存，查找所需记录的物理地址，然后再将该记录的页块送入内存。若索引表不大，则可将索引表一次读入内存。因此在索引文件中进行检索只需两次访问外存：一次读索引，一次读记录。

索引文件插入时，将插入记录置于数据区的末尾，并在索引表中插入索引项；删除时，删去相应的索引项；若要修改主关键字，则必须同时修改索引表。

当索引表很大时，一个页块容纳不下，查阅索引仍要多次访问外存。因此，可以对索引表建立一个索引，称为查找表。

多级索引是一种静态索引，各级索引均为顺序表，修改不方便，每次修改都要重组索引。因此，当数据文件在使用过程中记录变动较多时，应采用动态索引。

建立索引文件主要是如何组织文件的索引，特别是多级索引时需要建立 M 分查找树，一般主要使用 B-树和 B+树。

B-树和 B+树的操作主要是对 B-树和 B+树的结点的插入、删除和查找。值得注意的是：B-树和 B+树都是平衡树；B-树中的结点值都是关键字，而 B+树中的关键字都在叶结点上，即所有的叶结点都是关键字，内部结点都是临时结点。

2. 索引顺序文件

索引非顺序文件适合于随机存取，不适合于顺序存取。索引顺序文件既适合于随机存取，又适合于顺序存取。索引顺序文件是稀疏索引，占用空间较少。而索引非顺序文件是稠密索引。

(1) ISAM(索引顺序存取方法)

ISAM 是一种专为磁盘存取文件设计的文件组织形式，采用静态索引结构。

ISAM 文件由多级主索引、柱面索引、磁道索引和主文件组成(如图 10-2 所示)。

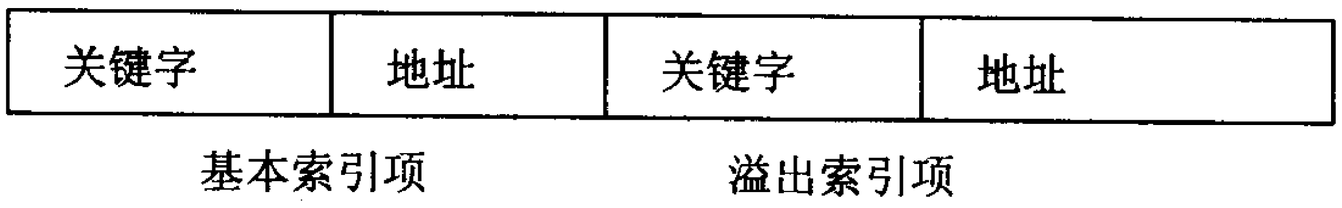


图 10-2　磁道索引项结构

文件的记录在同一盘组上存放时，应先集中放在一个柱面上，然后顺序存放在相邻的柱面上；对同一柱面则应按盘面的次序顺序存放。

为提高检索速率，通常可让主索引常驻内存，并将柱面索引放在数据文件所占空间居中位置的柱面上，目的是使磁头移动距离的平均值最小。

当插入新记录时，首先找到它应插入的磁道。若该磁道不满，则将新记录插入该磁道的适当位置上即可；若该磁道已满，则新记录或者插在该磁道上，或者直接插入到该磁道的溢出链表上。插入后，可能要修改磁道索引中的基本索引项和溢出索引项。

删除记录时，只要找到待删除的记录，在其存储位置上作删除标记即可，不需要移动记录或改变指针。

通常需要定期整理 ISAM 文件，因为经常对文件记录的删、增，造成大量记录进入数据溢出区，而基本区中的空间没有充分利用，造成基本区的空间浪费。

(2) VSAM

VSAM(虚拟存储存取方法)是一种索引顺序文件的组织方式，一般采用 B+树作为动态索引结构。

B+树是一种常用于文件组织的 B-树的变种树。一棵 m 阶的 B+树和 m 阶的 B-树的差异是：

- 有 k 个孩子的结点必有 k 个关键字；

- 所有的叶结点，包含了全部关键字的信息及指向相应记录的指针，且叶子结点本身依照关键字的大小，从小到大顺序链接；
- 上面各层结点中的关键字，均是下一层相应结点中最大关键字的复写(当然也可采用“最小关键字复写”的原则)，因此，所有非叶结点可看作是索引部分；
- B+的查找必须查找到叶结点为止，因为非末端结点只是起到了分界作用，其本身不是关键字。

可以对 B+树进行两种查找运算：一种是从最小关键字起进行顺序查找；另一种是从根结点开始进行随机查找。

在 B+树上进行随机查找、插入和删除的过程与 B-树的类似。只是在查找时，若非叶结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。B+树查找的分析类似于 B-树。

B+树的插入也仅在叶子结点进行，当结点中的关键字个数大于 m 时要分裂成两个结点，它们所含关键字的个数分别为$(m+1)/2$ 和$(m+1)/2$，并且它们的双亲结点中应同时包含这两个结点的最大关键字。

B+树的删除仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于 $m/2$ 时，则可能要和该结点的兄弟结点合并，合并过程和 B-树类似。

B+树每个叶子结点中的关键字对应一个记录，适宜稠密索引。若让叶结点中的关键字对应一个页块，B+树可用来作为稀疏索引。

VSAM 文件中没有溢出区，解决插入的方法是初建文件时留出空间。当插入新记录时，大多数的新记录能插入到相应的控制区间内。所以在 B+树中，有时需要内存和外存进行交换才能完成所需要的操作。

值得注意的是：为保持区间内记录的关键字从小到大有序，需要将区间内关键字大于插入记录关键字的记录向控制信息的方向移动。若干记录插入之后控制区间已满，则在下一个记录插入时，要进行控制区间的分裂，并修改顺序集中相应索引。通常控制区域较大，很少发生分裂的情况。

在 VSAM 文件中删除记录时，需将同一控制区间中，比删除记录关键字大的记录向前移动，把空间留给以后插入的新记录。若整个控制区间变空，则回收作空闲区间用，且需删除顺序集中相应的索引项。

和 ISAM 文件相比，基于 B+树的 VSAM 文件有如下优点：

- 较高的查找效率，查找一个后插入记录和查找一个原有记录具有相同的速度；
- 动态地分配和释放存储空间，而且不必对文件进行再组织。

基于 B+树的 VSAM 文件，通常被作为大型索引顺序文件的标准组织。

10.2.3 直接存取文件(散列文件)

散列文件是利用散列存储方式组织的文件。它类似于散列表，即根据文件中关键字的特点，设计一个散列函数和处理冲突的方法，将记录存储到存储设备上。通过散列函数和关键字直接计算记录的地址，所以亦称为直接存取文件。

与散列表不同的是，对于文件来说，磁盘上的文件记录通常是成组存放的，若干个记录组成一个存储单位，在散列文件中，这个存储单位叫做桶。假如一个桶能存放 m 个记录，则当桶中已有 m 个同义词的记录时，存放第 $m+1$ 个同义词会发生“溢出”。处理溢出虽可采用散列表中处理冲突的各种方法，但对于散列文件，一般主要采用拉链法。其处理冲突的步骤是：当发生“溢出”时，需要将第 $m+1$ 个同义词存放到另一个桶中，通常称此桶为“溢出桶”。前 m 个同义词存放的桶为“基桶”，溢出桶和基桶大小相同。相互之间用指针相链接。当基桶中没有找到待查记录时，就沿着指针到所指溢出桶中进行查找。因此，为提高查找速度，同一散列地址的溢出桶和基桶在磁盘上的物理位置不要相距大远，最好在同一柱面上。

在散列文件中删去一个记录时，一般仅需对被删记录作删除标记即可。

散列文件的优点是：文件随机存放，记录不需进行排序；插入、删除方便；存取速度快；不需要索引区，节省存储空间。

散列文件的缺点是：不能进行顺序存取，只能按关键字随机存取，询问方式简单，大量增删后，需要重新组织文件。

10.2.4 多关键字文件

1. 多重表文件

多重表文件是将索引方法和链接方法相结合的一种组织方式，它对每个需要查询的次关键字建立一个索引，同时将具有相同次关键字的记录链接成一个链表，并将此链表的头指针、链表长度及次关键字作为索引表的一个索引项。

例如：工资表文件，如表 10-1 所示。

表 10-1 工 资 表

编号	姓名	职称	基本工资	津贴	扣除	工资链	职称链
1001	张明	工程师	1000	100	50	1005	1005
1002	李虎	助工	800	50	20	1007	1007
1003	王卫	高工	1500	300	100	1006	1006
1004	李强	实习	600	30	30	Null	Null
1005	王娟	工程师	1000	150	80	1009	1009

(续表)

编号	姓名	职称	基本工资	津贴	扣除	工资链	职称链
1006	赵法	高工	1500	300	100	Null	Null
1007	李敢	助工	800	80	30	1008	1008
1008	崔五	助工	800	80	30	Null	Null
1009	张扬	工程师	1000	200	80	1010	1010
1010	张言	工程师	1000	200	80	Null	Null

由以上的例子可以看出，多重链表文件在建立时是比较复杂的，同时修改删除时，也同时需要修改相应的链表。所以多重链表比较适应于对已经建立链表的项进行查找，适合于静态查找。

2. 倒排文件

倒排文件和多重链表的区别在于次关键字索引的结构不同，倒排文件的次关键字索引称做倒排表。具有相同次关键字的记录之间不进行链接，而是在倒排表中列出具有该次关键字记录的物理地址。

例如：工资表文件。

数据文件如表 10-2 所示。

表 10-2　数据文件表

编号	姓名	职称	基本工资	津贴	扣除
1001	张明	工程师	1000	100	50
1002	李虎	助工	800	50	20
1003	王卫	高工	1500	300	100
1004	李强	实习	600	30	30
1005	王娟	工程师	1000	150	80
1006	赵法	高工	1500	300	100
1007	李敢	助工	800	80	30
1008	崔五	助工	800	80	30
1009	张扬	工程师	1000	200	80
1010	张言	工程师	1000	200	80

倒排表的工资链为：

600	1004
800	1002、1007、1008
1000	1001、1005、1009、1010
1500	1003、1006

职称链为：

实习	1004
助工	1002、1007、1008
工程师	1001、1005、1009、1010
高工	1003、1006

优点：可在倒排表中先完成查询的交、并等逻辑运算，得到结果后再对记录进行存取；存储具有相对独立性。

缺点：存取速度慢，同时不便于插入、删除。

10.3　文件的存储

文件的存储结构是指文件在外存上的组织形式。基本组织形式分为顺序组织、索引组织、散列组织和链组织。

计算机的存储设备有主存储器和辅助存储器之分，文件一般存储在辅助存储器中，所以对文件的存储主要考虑辅助存储器。辅助存储器的存储介质一般有磁带、磁盘和光盘等。考虑辅助存储器的主要原因是辅助存储器和主存储器相比主要有两个优点：

(1) 辅助存储器存储的文件是永久的，也就是当电源断电后，数据永久性地保存在辅助存储器中，而主存储器(RAM)中存储的内容会因断电而丢失；

(2) 辅助存储器可以方便地把磁盘等存储介质拿到其他计算机上使用，为计算机之间的信息传递提供了方便，而主存储器则不能。

辅助存储器尽管有以上两个主要优点并且价格便宜，但其访问时间和 RAM 相比比较长，这也是其很大的缺点。

如何提高辅助存储器的访问速度是文件存储时首要考虑的问题。存储介质的基本存取速度一般是固定的，所以要想提高速度应该遵循的基本原则是，尽量减少磁盘的访问次数。

减少磁盘的访问次数的方法一般有两种：

(1) 将信息安排在适当位置，目的是以尽可能少的访问次数得到需要的数据。文件结构的组织应使磁盘的访问次数最少。

(2) 压缩存储在磁盘中的信息，目的是在时间相同的情况下，获得更多的信息。这种方法在解压时需要占用 CPU 的时间，相对而言，比从磁盘中读取信息省下来的时间少得多。

10.3.1　磁盘

磁盘通常称为直接访问存储设备。磁盘是随机存取设备，磁盘中通常以存储非顺序文件为主，磁盘的读取单位不是以位为单位，而是以扇区为单位。使用磁盘时主要有 3 个时

间需要考虑：

(1) 寻找时间　磁头从当前位置移到数据存放磁道位置后所花的时间，或者叫做寻道时间。

(2) 延迟时间　磁头从当前扇区移动到数据存放扇区位置时所花的时间，或者叫做旋转时间。

(3) 传送时间　数据从磁盘读取数据，并将数据传送到内存的时间。

其中寻找时间和延迟时间可以通过操作系统的磁盘调度实现，以尽量减少数据的存取时间，而传送时间因为每个磁盘的旋转速率是固定的，所以对某个磁盘而言，其传输时间通常不变。

磁盘盘片组织如图 10-3 所示。

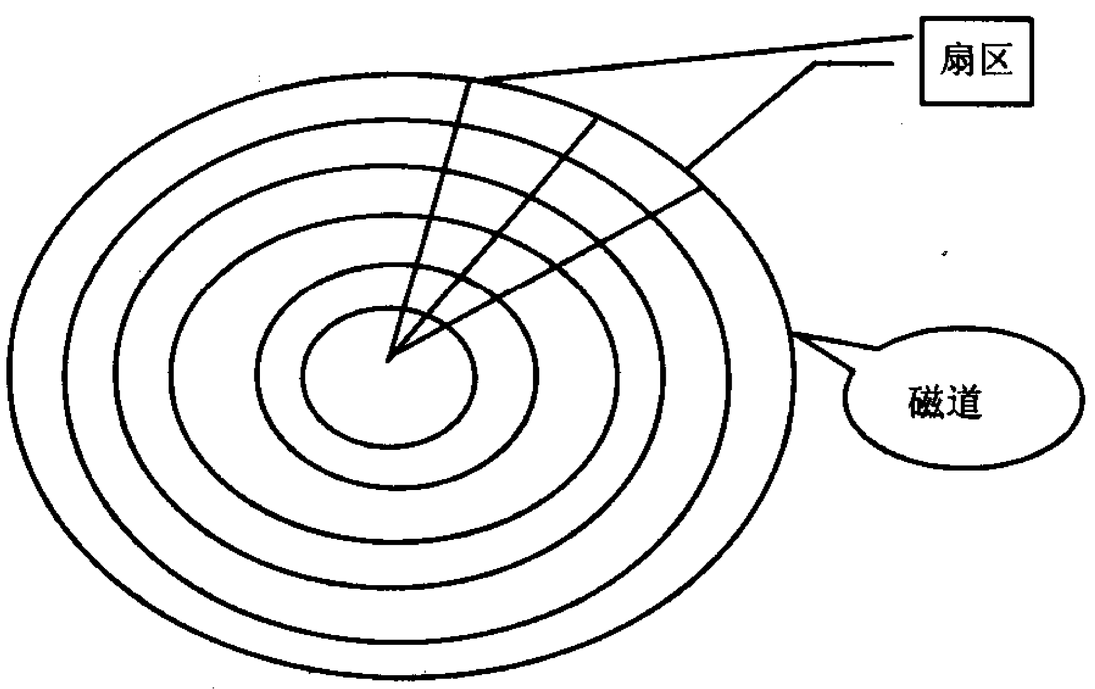


图 10-3　磁盘盘片组织

磁道是盘片上的同心圆。柱面是同一盘组上半径相同的磁道合在一起。扇区是将磁盘中的磁道划分成大小相同的扇面。

磁盘中存储的数据有固定的物理地址，磁盘的物理地址由 3 部分决定：柱面、磁道和扇区。

在现在的磁盘管理中，一般使用逻辑的组织管理单位——簇，簇一般由几个扇区组成，计算机在数据读取时，以簇为单位不再以扇区为单位，簇不能太大也不能太小。因为簇具有一个比较明显的特点：独占性。所谓独占性是指一个簇中只要存储了同一个文件的一个字节，就不能再存储其他文件的内容。所以簇太大会造成存储空间的大量浪费，而太小则使计算机管理的磁盘空间减少。

10.3.2　磁带

磁带通常称为顺序存储设备。磁带是顺序存取设备，使用磁带时和磁盘不同，它只能顺序访问。磁带如果不正转或者反转，就不能从当前位置到达目标位置。其组织如图 10-4 所示。

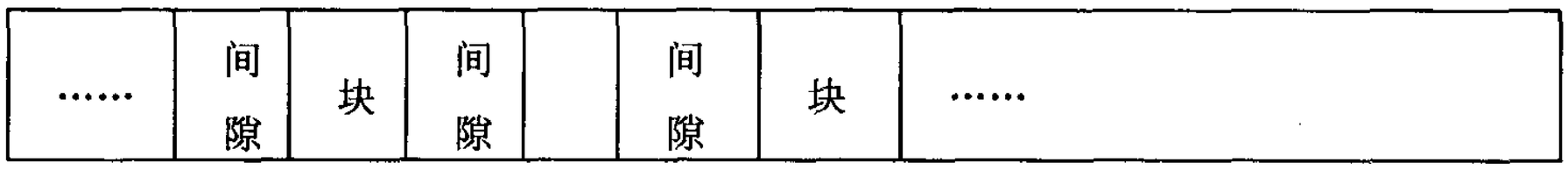


图 10-4　磁带组织图

磁带的存取单位是磁带长度，每一个存储单位是块，影响磁带的一个物理特性是磁带驱动器需要花费时间使磁带停下来，这就需要磁带的块间间隔。I/O 磁头需要识别间隔，同时间隔也浪费大量空间。间隔太大浪费空间，间隔太小 I/O 磁头的识别时间不够。一般将多条记录组织到一个块中，一个块中的记录的个数称为块因子。

磁带的另一个性能度量是数据的传送速度，块间间隔减少了数据的存储密度，提高了传输的速率。

磁带只适用于顺序访问，一般不能存储存取速度要求比较高的数据，它价格比较便宜，通常用来备份数据或归档。

对于磁带来说，它所需要的存取设备是磁带机，数据存储时通常可用多台磁带机并行工作，以提高效率。

对于外存来说：磁带只适宜存储顺序文件，磁盘则适宜存储顺序、索引、散列和多关键字等随机存取文件。

除了磁盘、磁带等外存设备以外，还有 CD-ROM、闪存等设备。

思考和练习

(1) 假设一个磁盘分成 10 个盘面，每个盘面有 128 个磁道，每个磁道有 64 个扇区，每个扇区有 512 个字节，问磁盘的容量是多少？

(2) 有一个 6250bpi 的磁带驱动器，记录大小是 160 个字节，一个块间间隔是 0.3 英寸，如果让 90%的磁带都包含数据，需要的块因子是多少？

(3) 对 9 个归并段和 8 个归并段的两个文件，试用 4 台磁带机做二路归并和多路归并，并对两种归并方法进行比较。

(4) 假设某个文件经过内部排序得到 100 个初始归并段，问：若要使多路归并 3 趟完成排序，则应取的归并路数应是多少？

(5) 假设 490 个初始归并段，将做 5 路磁带多路归并，问：需要多少步归并才能得到一个有序文件？初始归并段应在 5 台磁带机上如何分布？

参 考 文 献

1. 严蔚敏，吴为民. 数据结构. 北京：清华大学出版社，2003.7

2. 唐发根. 数据结构. 北京：科学出版社，2000.8

3. 黄刘生. 数据结构. 北京：经济科学出版社，1999.5

4. 黄国瑜，叶乃菁. 数据结构.北京：清华大学出版社，2002.9

5. 苏洋. Java 语言实用教程. 北京：希望电子出版社，2003.1

6. Clifford A.Shaffer 著. 张铭，刘晓丹译. 数据结构与算法分析. 北京：电子工业出版社，2001.2

7. 孙燕. Java 2 入门与实例教程. 北京：中国铁道出版社，2003.2

8. 张尧学，史美林. 计算机操作系统教程. 北京：清华大学出版社，2003.7

[G e n e r a l I n f o r m a t i o n]
□□=□□□□□□□□□□ □J a v a □□
□□=□□□□□
□□=2 3 4
S S □=1 1 4 5 1 1 7 2
□□□□=2 0 0 5 □ 0 7 □□ 1 □